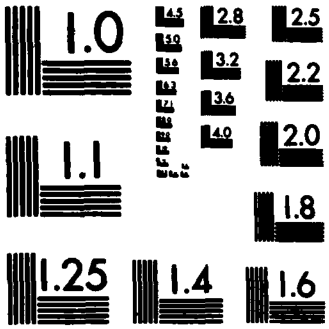
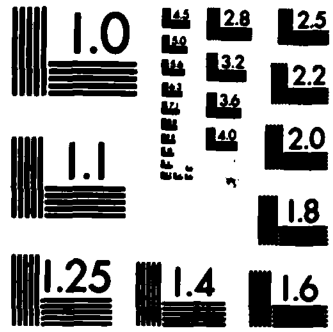
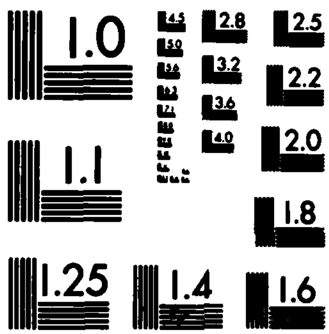
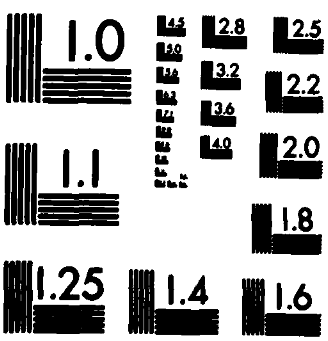
AD-A120 472 JOVIAL (J73) TO ADA TRANSLATOR(U) PROPRIETARY SOFTWARE INC LOS ANGELES CA M J NEIMAN JUN 82 RADC-TR-82-175 F30602-81-C-0217 1/2

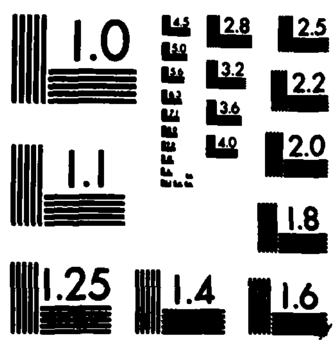
UNCLASSIFIED F/G 9/2 NL


1.0 1.1 1.25 1.4 1.6 1.8 2.0 2.2 2.5 2.8 3.2 3.6 4.0


1.0 1.1 1.25 1.4 1.6 1.8 2.0 2.2 2.5 2.8 3.2 3.6 4.0


1.0 1.1 1.25 1.4 1.6 1.8 2.0 2.2 2.5 2.8 3.2 3.6 4.0


1.0 1.1 1.25 1.4 1.6 1.8 2.0 2.2 2.5 2.8 3.2 3.6 4.0


1.0 1.1 1.25 1.4 1.6 1.8 2.0 2.2 2.5 2.8 3.2 3.6 4.0

(14)

RADC-TR-82-175
Final Technical Report
June 1982

JOVIAL (J73) TO ADA TRANSLATOR

Proprietary Software Systems, Inc.

Mark Neiman

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-82-175	2. GOVT ACCESSION NO. AD-A120 472	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) JOVIAL (J73) TO ADA TRANSLATOR		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report Jun 1981 - March 1982
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Mark Neiman		8. CONTRACT OR GRANT NUMBER(s) F30602-81-C-0217
9. PERFORMING ORGANIZATION NAME AND ADDRESS Proprietary Software Systems, Inc. 9911 West Pico Blvd, Penthouse K Los Angeles CA 90035		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55811917
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441		12. REPORT DATE June 1982
		13. NUMBER OF PAGES 122
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Elizabeth S. Kean (COES)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

JOVIAL
Ada
Translator
Compiler
Front-end

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document contains the functional description and system/subsystem specifications for a JOVIAL J73/Ada translator, and guidelines for J73 programmers who anticipate their programs will be converted to Ada at a later date. The functional description specifies the maximum JOVIAL J73 subset that can be converted to Ada. Techniques for the optimum automatic translation of the source code are specified. The J73 constructs that cannot be automatically translated are identified. The system/subsystem

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

specification provides a more detailed breakdown of the proposed translator.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

FUNCTIONAL DESCRIPTION

for the

JOVIAL (J73) TO ADA TRANSLATOR

Prepared by:

Mark J. Neiman

TABLE OF CONTENTS

Table of Contents - Continued

SECTION 1. GENERAL

1.1 PURPOSE OF FUNCTIONAL DESCRIPTION

This Functional Description for the JOVIAL (J73) to Ada Translator Investigation (F30602-81-C-0217) is written to provide:

a. The system requirements to be satisfied which will serve as a basis for mutual understanding between potential users and developers of a J73 to Ada Translator.

b. Information on performance requirements, preliminary design, user impacts, and costs.

c. A basis for the development of system tests.

1.2 Project References

a. Statement of Work, contract F30602-81-C-0127, RADC/PSS, June 1981.

b. Technical Proposal in response to RFP F30602-81-R-0058, RADC/PSS, January 1981.

c. "Translation of CMS-2 Programs to Ada," by Gilman, Crocker, Taylor, USC Information Sciences Inst., February 1980.

d. "Source-to-Source Translation: Ada to Pascal and Pascal to Ada," ACM SIGPLAN Symposium Proceedings, 1980.

e. MIL-STD-1815 - Reference Manual for the Ada Programming Language, December 1980.

f. Programming_in_Ada, by J. G. P. Barnes, 1981.

g. Programming__with__Ada:__An__Introduction__by__Means__of Graduated_Examples, by P. Wegner, 1982.

h. "Self-Assessment Procedure VIII," ACM Comm. Vol. 24, no. 10, by P. Wegner.

i. Computer, June, 1981.

j. "Programming Manual for JOVIAL (J73)", Softech.

k. "Software Engineering Exchange," IBM FSD, October 1980, Vol. 3, no. 1.

l. "Rationale for Design of the Ada Language," ACM SIGPLAN Notices, Vol. 14, no. 6, Part B.

m. DIANA_Reference_Manual, by Goos & Wulf, March 1981.

n. Computer Program Development Specifications, by T.I. and Intermetrics (Compiler/Environment), March 1981 Drafts.

o. MIL-STD-1589B - JOVIAL (J73).

p. F.C.S.C. Conversion Products/Aids Survey, Report GSA/FCSC-81/004.

q. JOVIAL___(J73)__to__Ada__Translator__System, by R. L. Brozovic, Master's Thesis (AFIT), December 1980.

1.3 Terms and Abbreviations

The following terms and abbreviations will be used throughout this Functional Description:

Erroneous	A high order language program which contains one or more violations of language semantics which are not detected by a compiler. Erroneous programs have unpredictable run-time results.
External	A program element that is referenced by modules which are compiled separately from the module in which the element is declared.
J73	The programming language JOVIAL (J73) as specified by MIL-STD-1589B.
Module	A portion of a J73 or Ada program which is logically distinct from the rest of its program and which may be compiled or translated separately.
Program	All of the modules of a J73 or Ada program, as opposed to an individual compilation unit.
TPF	Translation Parameter File - a user accessible file which specifies which translation options will be used for a run of the Translator.
Translator	The proposed JOVIAL (J73) to Ada translator.

SECTION 2. SYSTEM SUMMARY

2.1 Background

The Department of Defense is currently engaged in a long term effort to define and develop a new high order programming language, Ada. Ada is to be used as the standard implementation tool for embedded computer systems. Standardized and validated Ada compilers and environments will not be available for another year or two; moreover, many new and ongoing Air Force projects are using the present standard language, JOVIAL (J73), for implementation of real-time software systems. The need for a JOVIAL (J73) to Ada Translator is driven by two major problems:

a. Existing software written in J73 will eventually be used in Ada-based systems.

b. Software developed during the Ada development period (1980-1984) cannot use Ada exclusively and therefore must be written in J73.

A Translator would enable J73 programs to be converted to the new standard language so that the advantages of maintainability and universality of Ada may be exploited. The translator will be needed not only during the Ada development period, but also afterward; future maintainers of embedded systems will be experts in the use of Ada and the Ada Programming Support Environment (APSE) rather than JOVIAL. These maintainers will benefit greatly from having a J73 to Ada Translator as a part of the APSE.

2.2 Objectives

The objective of the proposed JOVIAL (J73) to Ada Translator is the automatic translation of J73 source programs to equivalent Ada source programs. Because some J73 constructs cannot be automatically translated to Ada, the Translator must detect and flag any constructs which it does not translate.

Another goal of the Translator is flexibility. A number of J73 constructs have two (or many) possible Ada translations. The user may wish to control some or all of the choices made by the Translator. This will require parameterization of the translation options, with a user-accessible file for the parameter values.

pical J73 program consists of many separately compiled modules share data specifications using the DEF, REF, and COMPOOL ucts. Translation of individual J73 modules into Ada modules preserve the semantics of J73 compools and externals, while ting in well structured Ada programs, is a major design tive.

ugh the Translator is designed as a stand-alone product, it is ioned as a part of the Ada Programming Support Environment. With help of compilers, text editors, file managers, and other APSE , the Translator will provide significant (though not total) ation of the conversion of J73 programs for use in the Ada onment.

Existing Methods and Procedures

are no existing implementations of a Translator which satisfy objectives described in the preceding paragraph. Many automatic lators exist for simpler languages such as COBOL, FORTRAN, RPG, numerous assembly languages, but at present, the only method of ving production quality translation of J73 to Ada is manual lation.

se Ada compilers and environments do not yet exist in complete, ated implementations, one may assume that very little manual lation of J73 to Ada has been performed. However, if manual ation is to be performed, two major ingredients are required. First is a set of rules, which may be as formal as a language specification or as informal as a set of rules and procedures, specify the mapping of J73 onto Ada. The second is a programmer oup of programmers who are experts in both languages. While the l ingredient is somewhat rare, the first is probably nonexistent. lation of large programs (i.e., several hundred modules) would be itively expensive, even if both ingredients were acquired. er, a manual translation of a large realtime system would be t to much human error and inconsistency; the final product would e "flyable" without very extensive redesign to remove the table translation errors.

Proposed Methods and Procedures

Translator proposed in this document is intended to automate the ation of J73 to Ada to the largest extent practical. Although automation may be impossible, it is anticipated that 80-95% of fort involved in manual translation will be removed. In addition anslating nearly all of a J73 program to equivalent Ada, the ator will detect untranslatable code and will generate stub s for additional Ada code required by the translated program.

2.4.1 Summary of Improvements

The improvements of automatic translation over manual translation are summarized below:

a. Vastly increased throughput.

b. Greater consistency.

c. The Translator will be quite thorough, either translating or flagging everything in the J73 program. A human translator might inadvertantly omit a part of the program.

d. Greater flexibility. If a certain aspect of the translation appears unacceptable, a different translation could be obtained by changing parameters and re-running the Translator. This would be unfeasable for manual translation.

e. Reduced cost. The cost of automatic translation of J73 modules would approximately equal the cost of compiling the modules. The additional costs involved in completing the translation of a program (manually translating or reprogramming the untranslatable portions) would be much less than the time and money saved by automating the bulk of the translation process.

2.4.2 Summary of Impacts

Since there are no ongoing J73 to Ada translation projects, there are assumed to be no impacts on equipment, software, organizations, or operations. The Translator would be developed on an existing medium-to-large scale computer system, and its installation would be similar to that of any other stand-alone software system.

2.5 Assumptions and Constraints

The Translator described in this document is assumed to process correct J73 programs and to output correct Ada programs. The sense in which the input and output are considered to be "correct" is discussed in paragraphs 3.1.1 and 3.3.1.2.

SECTION 3. DETAILED CHARACTERISTICS

3.1 Specific Performance Requirements

This paragraph describes the performance requirements to be satisfied by the Translator with regard to accuracy, validity, timing and capacity.

3.1.1 Accuracy and Validity

The translations performed by the Translator will be accurate in the sense that the resulting Ada programs will be semantically equivalent to the J73 programs from which they were derived to the largest extent possible. Except for certain untranslated constructs, which will be clearly flagged in the output, the Ada produced by the Translator will be a valid Ada program in that

a. It will contain no syntax errors;

b. Any missing code that is required for execution of the program will be clearly identified;

c. It will be compilable in a standard Ada environment without modifications (such as reorganizing statements and declarations or renaming modules or variables);

d. It will conform to general standards for readable, well structured programming.

In general, two versions of a program cannot be guaranteed to have absolutely identical run-time behavior in two different environments, even if the versions were generated from the same source code (e.g., a J73 program compiled for two different targets). Therefore, the Translator cannot be required to produce a "perfect" translation of a non-trivial program. However, it will be required to preserve the original program semantics wherever possible, at the expense of some run-time efficiency if necessary, and to inform the user of any possible deviations from J73 semantics that are introduced by the translation.

3.1.2 Timing and Capacity

Although portions of a program may require repeated translation to resolve various translation problems, the overall translation process will be a one-time task. High performance with respect to throughput is, therefore, not given a high priority. The Translator should process J73 source code at about the same speed as a compiler, roughly 100-1000 source lines per CPU second on a typical mainframe host system.

The Translator and its host environment must have the capacity to store and process an entire J73 program at one time. This typically means a capacity on the order of 1000 modules of 50-200 source lines each for a large flight software system.

3.2 System Functions

This paragraph describes the specific functions to be performed by the Translator. These functions can be considered to comprise an informal specification of a mapping of the J73 language onto the Ada language. The mapping is described in six parts:

a. Program Structure
b. Types and Object Declarations
c. Executable Constructs
d. Directives
e. Intrinsic Functions
f. Miscellaneous Issues

The following subparagraphs describe each of these functions by providing a rationale for the particular mappings selected.

3.2.1 Program Structure

This subparagraph describes translation functions which are related to program structure, including modules, externals, and procedure specifications.

3.2.1.1 Modularity: Compools and Packages

J73 programs are written as separately compiled modules. Typically, an individual module will consist of either a compool or a small procedure, either of which may include external compool references. The Translator must be compatible with this kind of program structure, permitting both global compool references and efficient translation of small modules. These goals are not obviously in agreement: global data referencing implies knowledge of many modules during a single module's compilation. The JOVIAL environment satisfies these goals by creating compool output files when compiling compools. Small modules which reference compools are compiled separately and efficiently by reading the (previously created) compool output files. Unfortunately, there is no "standard compool output file" format -- each compiler has its own private format. Satisfying the goals mentioned above, therefore, presents a major problem for the Translator.

Individual J73 compool modules will be translated to Ada package specifications. This will permit other modules to use the resources of separately compiled (and separately translated) compool modules using a WITH clause. For example, a compool module

```
START
COMPOOL globalvariables;
BEGIN
     "sequence of declarations"
END
TERM
```

will be translated to

```
PACKAGE globalvariables IS
   --  sequence of declarations
END globalvariables;
```

A compool directive

```
!COMPOOL globalvariables;
```

will become

```
WITH globalvariables;
```

making the sequence of declarations in "globalvariables" available to the module containing the WITH clause. This translation precisely reflects the J73 semantics of compool usage, including the order of compilation: the compool/package must be compiled before it may be referenced, and the content of the compool/package must be part of the compilation/translation environment. If these conditions are met, then the Translator will satisfy the goal of preserving a J73 program's modular structure while translating it to Ada.

A more detailed example will illustrate how this translation process handles multiple and partial compool use. Two compools

```
START
COMPOOL comp1;
BEGIN
     "declarations of variables AA, BB"
END
TERM

START
COMPOOL comp2;
BEGIN
     "declarations of variables CC,DD"
END
TERM
```

will become the packages

```
PACKAGE comp1 IS
   --  declarations of variables AA,BB
END comp1;

PACKAGE comp2 IS
   -- declarations of variables CC,DD
END comp2;
```

Another compilation unit which uses comp1 and part of comp2

```
!COMPOOL comp1;
!COMPOOL comp2 CC;
          .
          .
          .
     "references to AA,BB,CC"
```

will be translated to

```
WITH comp1, comp2;
USE comp1, comp2;
   -- references to AA,BB,CC
          .
          .
          .
```

The USE clause makes the variables declared in comp1 and comp2 directly visible in the Ada module. If the USE clause were omitted, those variables would need to be qualified with their corresponding package names:

```
WITH comp1, comp2
          .
          .
          .
   -- references to comp1.AA, comp1.BB, comp2.CC
```

The dotted notation will be used for names which are imported by a partial compool directive. This will avoid ambiguity in case the same name was declared (but not imported from) another compool. For names imported by a complete compool directive, there will be no ambiguity in regards to which package a variable belongs, and the dotted notation may be avoided by including a USE clause in the package or procedure specification.

A J73 program which contains declarations of the same name in more than one compool is erroneous. The J73 compiler will not necessarily detect this error (but the linker would). It is interesting to note that such a program, after being translated to Ada in the manner described above, would always be diagnosed at compile time. This is true because Ada requires disambiguation of all references during compilation, while J73 does not.

Another interesting aspect of this translation technique is that the distinction between a complete compool directive and a partial compool directive is removed. Partial compools are specified in J73 as an aid to efficient compilation; the compiler knows that it need not bother reading all of the compool file into its symbol table, and may read only what is needed. A module will have the same meaning as if it had requested the entire compool, but will compile faster and in less space. The Translator will ignore this distinction for two reasons:

(1) There is no straightforward Ada version of a partial compool directive - packages are used only in entirety.

(2) The Translator will model the Ada environment in the sense that it will have global knowledge of global and external objects during translation, and will not need to input compool data on a module-by-module basis. This is discussed further in paragraph 3.2.1.3.

3.2.1.2 Context-Dependent Declarations

J73 permits the programmer to make objects either static or externally visible on an explicit, declaration-by-declaration basis. Examples:

```
ITEM eternal STATIC S;
DEF ITEM external S;
```

These items are either statically allocated or externally visible regardless of the context in which the declarations appear. This concept does not exist in Ada. Objects are "allocated" or "externalized" in Ada according to context. A variable, for example, will be static only if it is declared outside of any kind of local structure, such as a procedure or function; it will be externally visible only if it is declared in a package specification or procedure specification. Translation of procedures containing explicit STATIC and DEF declarations, therefore, is really a program structure issue, and is discussed in the next two paragraphs.

3.2.1.2.1 Procedure Specification

In their simplest forms, J73 procedures and functions may be translated directly to Ada procedures and functions. For example, the J73 procedure

```
START
!COMPOOL compl;
DEF PROC proc1;
BEGIN
"local declarations and executable statements"
END
TERM
```

becomes the very similar Ada procedure

```
WITH compl;
PROCEDURE proc1 IS
BEGIN
  -- local declarations and executable statements
END proc1;
```

This straightforward translation is correct only if the "local declarations" included no instances of the J73 "DEF" or "STATIC" constructs. For example, if

```
START
DEF PROC proc2;
BEGIN
  ITEM xx STATIC S;
  "...the rest of the procedure"
END
TERM
```

were translated as in the preceding example, the variable xx could not be made static. Ada has no explicit construct for declaring local static data; anything declared inside a procedure body is implicitly automatic, existing only when the procedure is invoked. What is needed is an Ada structure which provides locality (hiding the declaration from other procedures) while also providing permanent existence for the data being declared. This is accomplished by the Ada package; declaring the variable inside a package body but outside the procedure body will make the variable static and local. The only complication is the name of the package. The procedure "proc2" may be translated to

```
PACKAGE proc2_pack IS
     PROCEDURE proc2; -- specification of proc2
END proc2_pack; -- end of package specification

PACKAGE BODY proc2_pack IS
     xx:INTEGER; -- inside package body but outside proc body
     PROCEDURE proc2 IS -- body of proc2
     BEGIN
       -- the rest of the procedure
     END proc2;
END proc2_pack; -- end of package body
```

In this translation, xx is local to the package proc2_pack, but since proc2_pack contains nothing but the procedure proc2, xx is effectively local to proc2. However, the declaration of xx outside of the procedure body ensures that storage will be allocated for the life of the package, rather than merely for the life of an invocation of proc2. Since packages are inherently static, xx will be static. (Note: an Ada construct that is inherently dynamic rather than static is the task. This construct does not appear to be necessary for representation of J73 programs.) The package makes the name "proc2" visible to other compilation units by including a specification of proc2 in the package specification. Both xx and the body of proc2 are hidden from other compilation units, preserving the semantics of the original J73 version.

The "overhead" involved in the creation of a package for a procedure with static local data is further justified by the fact that the package structure solves another major translation problem - that of external declarations, discussed in the next paragraph.

The main program module will be translated to a procedure or a package using the same techniques as for an ordinary procedure module; Ada does not require a syntactic distinction between main and subordinate modules. Procedures and functions may be nested in Ada, just as in J73, with no change in program semantics. All Ada subprograms are compiled to be reentrant and recursive, so that the Translator may ignore the RENT and REC attributes in a procedure declaration.

A module containing multiple DEF PROC's (i.e., non-nested procedures and functions) will be translated to a package which contains multiple procedure or package declarations. For example, a module such as

```
START
"declarations global to AAA and BBB"
DEF PROC AAA; "procedure with some DEF items"
BEGIN
   .
   .
   .
END
DEF PROC BBB; "procedure with no DEF or static data"
BEGIN
   .
   .
   .
END
TERM
```

would become a package specification whose name is derived from the two DEF PROC's:

```
PACKAGE AAA_BBB_pack IS
  declarations global to AAA and BBB
     PACKAGE AAA_pack IS
          PROCEDURE AAA...
            .
            .
            .
     END AAA_pack;
     PROCEDURE BBB IS
            .
            .
            .
     END BBB;
  END BBB;
END AAA_BBB_pack;
```

Machine specific functions and procedures are not coded in J73, and therefore will not be processed by the Translator. The user may code machine specific routines in Ada using the technique described in Section 13.8 of the Ada Reference Manual.

3.2.1.2.2 Externals

The J73 REF and DEF constructs specify declarations that are used externally. As previously stated, Ada externals must be declared in a package specification. The only Ada construct that resembles the J73 REF is the WITH clause, which was shown to be equivalent to the J73 compool directive. The WITH clause may be used to translate J73 REF declarations in a similar manner.

A J73 module containing one or more declarations that are to be made available to external modules (e.g., DEF ITEM, DEF TABLE, etc.) will be translated into an Ada package. The specification of the package will contain all declarations which are DEF'ed in the J73 version. If the module is a DEF PROC, the package specification will include a declaration of the procedure itself. If the module is a compool, in which everything is DEF'ed, we have the situation discussed in 3.2.1.1: the whole module becomes a package specification with no body.

With all DEF'ed objects declared in package specifications, other modules can REF the objects by using a WITH clause. In general, a J73 procedure of the form

```
START
DEF PROC procname;
BEGIN
   "local declarations which are DEF'ed"
   "other local declarations"
   "the rest of the procedure body"
END
TERM
```

will be translated to an Ada package of the form

```
PACKAGE procname_pack IS
   PROCEDURE procname;
   -- local declarations which were DEF'ed
END procname_pack; -- end of package specification

PACKAGE BODY procname_pack IS
   -- static local declarations
   PROCEDURE procname IS
     BEGIN
        -- remaining local declarations
        -- rest of procedure body
     END procname; -- end of procedure body
END procname_pack;  -- end of package body
```

so that all objects which were DEF'ed can be accessed externally (REF'ed) using "WITH procname_pack".

A problem arises when REF declarations are used in compools. REF PROC's, REF ITEMS, etc., are sometimes included in compools as a means of copying the REF declarations into other modules. If compool REF declarations were translated to WITH clauses, the resulting Ada program would contain many circular compilation dependencies. For example, a compool full of REF PROC's may be imported by all modules which contain procedure calls, resulting in

```
WITH P1, P2, P3...     -- REF's to each procedure
PACKAGE refproccompool IS
    .
    .
    .
END refproccompool;
```

for the compool and

```
WITH refproccompool;  -- imports the compool
PACKAGE PN_pack IS
    .
    .
    .
END IN_pack;
```

for each procedure. This is unacceptable, since the mutual WITH clauses preclude any possible order of compilation (*A module must be compiled after all modules whose names appear in its WITH clause*). This problem is solved by placing the REF PROC in the module that actually needs it, rather then in the compool from which the REF PROC is imported. Thus, a procedure which reads in a REF PROC from a compool will get a WITH clause for the REF PROC. For example, if procedure P22 reads in (from a compool) a REF of procedure P44, then P22 will get the "WITH P44" clause; the compool will not. In general, REF declarations in compools will result in WITH clauses for the compool itself only if the REF is to another compool; otherwise, the REF declarations will simply be removed from the compool and placed, in the form of WITH clauses, in modules which import the compool.

3.2.1.3 Summary

The functions performed by the Translator with respect to program/module structure are summarized below.

a. Compools will be translated to package specifications with no package body.

b. Compool directives will be translated to WITH clauses of the form "WITH compool-name".

c. Procedures and functions which contain no static or DEF declarations will be translated into Ada procedure and function bodies.

d. Procedures and functions which contain static or DEF declarations will be translated into Ada packages with the following characteristics:

- the name of the package will be of the form "procedure_name_pack".

- the package specification will contain all declarations which are DEF'ed.

- the package body will contain the procedure or function body, with non-static declarations inside the procedure (function) body and static declarations outside the procedure (function) body.

e. Modules containing REF declarations will be translated to modules that use WITH clauses to access externally DEF'ed objects. For each module whose external declarations are needed (REF'ed), a clause of the form "WITH name_of_package_containing_the_original_declaration" will be placed before the module heading (i.e., before the package or procedure or function declaration). This will remove the need for an explicit declaration in place of the REF: the declaration will be imported from the module that originally included it.

f. In the case of REF declarations in compools which refer to non-compool modules, the WITH clause generated by the REF will appear in the modules that import the compool, rather than in the compool-package itself.

A major implication of these functions is that the Translator must provide a mechanism for determining the global context of names. For example, the Translator must know in what package an object is DEF'ed in order to translate a REF of that object. This global knowledge of name context is analogous to the Ada environment itself. The J73 environment maintains global knowledge only of compool declarations: externals are not resolved until the compiled modules are linked. Creating a global data base during compilation/translation of the program involves some overhead in both time and space for the compiler/translator, but the extra resources required are considered worthwhile for two reasons:

a. There is no other way to translate J73 external references to cleanly-compilable Ada code.

b. The Ada programs resulting from the translation techniques described in this paragraph will not only be "correct" in the sense of accurately reflecting J73 semantics; they will also be "well-structured Ada," using the concepts of packaging, data hiding, and name visibility in precisely the manner that would be used by a good Ada programmer.

The efficient implementation of the global data base for name context determination is discussed in a later report.

3.2.2 Types and Declarations

This paragraph describes translation functions which are related to declaration and use of types, variables, and constants.

3.2.2.1 Predefined Types

Both J73 and Ada feature predefined types that may be used in a declaration along with range and precision specifiers. J73 syntax for numerical and string types feature a kind of shorthand notation, such as

```
TYPE unsigned U; "fullword unsigned integer"
TYPE halfint S 8; "eight bit signed integer"
ITEM sixchar C 6; "six byte character string"
ITEM wholefrac A 0,31; "fixed point number with no scale
                        bits and 31 fraction bits"
```

in which precision, range, or size of a type is given in terms of the number of bits or bytes needed to represent values of the type. In Ada, these attributes are specified in terms of explicit range constraints, fixed point "delta" and floating point "digits" for numerical types, and by arrays for string types. The J73 predefined types (U,S,A,F,C,B) will be translated to Ada type names such as

J73 TYPE NAME	ADA TYPE NAME
U	U_type
U 5	U5_type
S 31	S31_type
A 5,26	A5_26_type
A 14	A14__type
F 27	F27_type
C 10	C10_type
B 18	B18_type

on. For each unique Ada type name generated in this manner, the
tor will generate a declaration which will go into a package
"J73_predefined_package." The contents of this
tor-generated package will be output upon user request (see
).

tions in J73_predefined_package for integer types will be
translations of size to range. Examples:

```
TYPE    U_type IS INTEGER RANGE 0..INTEGER'LAST;
TYPE    U5_type IS INTEGER RANGE 0..31;
TYPE    S31_type IS INTEGER RANGE -2**31..(2**31)-1;
```

ng these types as subtypes of the predefined type INTEGER will
that implicit type conversion will be made between any two
types, as in J73. If new types were declared, rather than
s, implicit conversions would not occur: Ada treats distinctly
d types as non-matching types, even if the types are declared
ally.

oint types require a specification of "delta", the error bound,
is equal to 2**(-F) for a J73 fraction size of F. Thus, a
n size of 4 will yield a delta of 1/16; a fraction size of -8
eld a delta of 256. The range of a fixed point type is computed
ly to that of signed integers. Examples:

```
A5_26_type IS DELTA 1.0/2**26 RANGE -2**5..(2**5)-(1.0/2**26);
```

s better coded as

```
_5_26: CONSTANT := 1.0/2**26;
E A5_26_type IS DELTA del_5_26 RANGE -32..32-del_5_26;
```

example:

```
_14__: CONSTANT := 1.0/2**(WORD_LENGTH-15);
E A14__type IS DELTA del_14__ RANGE -2**14..2**14-del_14__;
```

e scale and fraction specifiers are handled in the same manner.
ype "A -6,37" will yield

```
6_37: CONSTANT: = 1.0/2**37;
An6_37 IS DELTA del_n6_37 RANGE -1.0/2**6..1.0/2**6-del_n6_37;
```

declares a fixed point fraction type whose values are between
nd (about) 1/64 with 31 bits of precision.

Unfortunately, there can be no predefined fixed point type from which all needed types can be "subtyped", as with integers. The reason for this is the rule that the values of all subtypes must be subsets of the values of the parent type. The values of all possible fixed point types are not subsets of any Ada-definable type. Therefore, fixed point types will be distinct types, and any J73 implicit type conversions will be translated to explicit Ada type conversions.

Floating point types require accuracy specification in terms of the number of decimal digits. For B bits of precision in the mantissa, the number of decimal digits needed for equal precision is

$$B/\log_2 10 \; *$$

Examples of floating point type declarations:

```
SUBTYPE F_type IS FLOAT;
SUBTYPE F27_type IS FLOAT DIGITS 8;
```

An Ada compiler will generate floating point code with at least the precision specified in the type declarations; this is identical to J73 semantics for floating point arithmetic. Implicit type conversions between objects of floating point types will work the same way as previously described for integer types.

All of the preceeding examples assume a two's-complement target machine. The range specification needed for integers and fixed point numbers would be different for a one's-complement (or sign-magnitude) target in that the lower bound is one "error bound" closer to zero. In general, for a precision or scale size of B bits, the lower bounds of signed integer and fixed point types are

	signed integer	fixed point
two's complement:	-(2**B)	-(2**B)
one's comp/sign mag:	-(2**B)+1	-(2**B)+delta

The upper range bounds are the same for either representation ((2**B-1) for signed integer, (2**B-delta) for fixed point). The Translator will select lower bounds based on a TPF entry for the desired target machine representation.

----

* The actual number of digits will, of course, be the least integer greater than this quantity.

J73 character types may be represented as Ada string types, such as

```
SUBTYPE C_10_type IS STRING(1..10);
SUBTYPE C_type IS STRING(1..1);
```

so that objects of character type will be accessible as arrays. This permits both access to the entire object and access to a substring of the object (using a slice: its name followed by a range specification), allowing straightfoward translations of J73 type conversions and byte operations.

The remaining J73 predefined type, bit type, is the most problematical. Ada includes a boolean type which corresponds to the J73 type, "B 1", but contains nothing equivalent to a bit string type. Two possible translations involve the use of integer types and array types.

Objects of integer type are unsuitable for representation of bit strings for two reasons. First, the maximum allowed size of an integer in a typical Ada implementation will be one or two target words (16-64 bits), while J73 bit strings may be dozens of target words in length. Thus, long bit strings such as "B 256" would be unmappable into Ada integers. The second problem involves boolean operations. Since Ada permits only boolean arguments to operators such as "and", "or", "not", and "xor", performing such operations on integers would require the equivalent of overloading of the operators for the types in question. Conversion of integer types to boolean or array types is illegal; the implementation of boolean operations on integers would be awkward and inefficient.

A workable translation of J73 bit types uses arrays of booleans. J73_predefined_package will include the declaration

```
TYPE bit_string IS ARRAY (INTEGER RANGE < >) OF BOOLEAN;
```

to establish a parent type for specific subtypes such as

```
SUBTYPE B_18_type IS bit_string (0..17);
SUBTYPE B256_type IS bit_string (0..255);
```

and, for consistency,

```
SUBTYPE B1_type IS bit_string (0..0);
```

This mapping will permit bit strings to be accessed in the same manner as character strings, using "slice" references for type conversions and substring operations (e.g., the J73 "bit" operator). The Ada boolean operators are directly applicable to boolean array types, so that no inefficiency will be incurred in performing boolean operations. The only remaining problem is storage efficiency; J73 bit strings are always packed, while Ada arrays are not. This problem is solved by including

```
PRAGMA PACK (bit_string);
```

in J73_predefined_package, which requests the Ada compiler to pack all arrays of type bit_string to minimize space.

3.2.2.2 Type and Object Declarations

Translation of type, variable, and constant declarations in J73 will be translated to Ada declarations using the predefined types discussed in the preceeding paragraph whenever possible. Declarations which cannot make use of the predefined types will use distinct type definitions as necessary. The following paragraphs discuss the translation of each kind of J73 type and object declaration in the order given:

a. Scalar (numeric, string, and enumeration) types
b. Tables
c. Pointers
d. Other (blocks, defines, etc.)

3.2.2.2.1 Scalar Types

Declarations of types and objects of numeric or string types will be translated using the predefined types declared in J73_predefined_package.

Examples:

```
ITEM speed U 10;
CONSTANT ITEM pi A 2,15 = 3.14159;
TYPE name C 13;
ITEM mask STATIC B 36 = 4B'800000000';
```

will be translated to

```
speed: U10_type;
pi: CONSTANT A2_15_type:=3.14159;
SUBTYPE name IS C13_type;
mask: B36_type:=(0=>TRUE, 1..35=>FALSE);
```

The translations of the first three of these declarations are straightfoward uses of types (subtypes) declared in J73_predefined_package. The fourth declaration involves two additional features: the STATIC specifier and a preset value. Translation of static declarations involves the context of the declarations, as described in 3.2.1.2.1. Translation of the preset of the bit string requires converting a J73 bit constant to a corresponding Ada aggregate. In this example, the J73 literal whose first bit is a "1" and whose remaining bits are "0" becomes an aggregate whose zero position has a value of TRUE and whose first through 35th positions have values of FALSE. This aggregate has the effect of initializing each component of the 36 component array, just as the literal, 4B'800000000', initialized each bit of the 36 bit item in the J73 version. The aggregate could be written equivalently as (0=>TRUE, OTHERS=>FALSE), with exactly the same effect.

Round-or-truncate attributes in numerical declarations will not affect the translation of the declarations themselves. However, conversions to integer and fixed point types, as well as assignments to floating point types will, if required, generate function calls to user supplied routines which will perform the desired rounding or truncation. These function calls may be suppressed using a TPF entry.

Enumeration types are easily translated. For example, the declarations

```
TYPE color STATUS (V(red), V(amber), V(green));
ITEM signal color;
CONSTANT ITEM stoplight color = V(red);
```

will be translated to

```
TYPE color IS (red, amber, green);
signal: color;
stoplight: CONSTANT color:=red;
```

Removal of the letter "V" and the parentheses from status constants may cause ambiguity in the resulting translation. Since other identifiers in the module containing the declaration of "color" may be spelled the same way as "red", "green", or "amber", dotted notation (e.g., color.red) will be used to translate references to these values.

Item declarations which include a status list definition, such as

```
ITEM condition STATUS (V(good), V(bad));
```

will be broken into two declarations:

```
TYPE condition_type IS (good,bad);
condition: condition_type;
```

This is necessary because an Ada object declaration must contain a type (subtype) name rather than a type (subtype) definition.

Status type declarations with specified representation attributes will be translated using Ada representation specifications. A declaration such as

```
TYPE points STATUS 3(1 V(pointafter), 2 V(safety),
                     3 V(fieldgoal), 6 V(touchdown));
```

will yield a basic type declaration and two representation specifications:

```
TYPE points IS (pointafter, safety, fieldgoal, touchdown);
FOR points'SIZE USE 3;
FOR points USE (pointafter=>1, safety=>2,
                fieldgoal=>3, touchdown=>6);
```

This technique will assure proper representation of values of the status type.

3.2.2.2.2 Tables

A J73 table is an aggregate data object. The simplest form of a table declaration includes a name, a dimension list, and an item type description, such as

```
TABLE employees (99) C 15;
```

which declares an array of 100 character string elements (indexed 0 through 99). This is equivalent to

```
employees: ARRAY (0..99) OF C15_type;
```

Tables bodies correspond to records. A serial table will be translated in two parts. First, a record type will be declared to match the table body. Second, an array of record type will be declared to match the table name and dimension list. For example, a table containing employee information declared as

```
TABLE employees (99);
BEGIN
     ITEM name C 15;
     ITEM rank ranktype; "ranktype is declared elsewhere"
     ITEM serialnumber U;
END
```

will be translated to

```
TYPE employees_type IS
RECORD          --declares the type of the table body
     name: C15_type;
     rank: ranktype;
     serialnumber: U_type;
END RECORD;

employees: ARRAY (0..99) OF employees_type;   --declares the table
```

The translation is done in two parts because an Ada array declaration must use a type name rather than a type description. Tables with more than one dimension will become arrays of more than one dimension:

```
TABLE multidim (22, 14:114, 511)...
```

becomes

```
multidim: ARRAY (0..22,14..114,0..511)...
```

Packing specifiers, words-per-entry, and location specifiers will be translated by means of representation specifications. If the table "employees" were declared as a specified table,

```
TABLE employees (99) W 6;
BEGIN
     ITEM name C 15 POS(8,0);
     ITEM rank ranktype POS(0,4);
     ITEM serialnumber U POS(1,5);
END
```

its translation will consist of the record and array declarations given earlier and the representation specification

```
FOR employees USE
RECORD AT MOD 6*word;     --six words per entry
     name AT 0*word RANGE 8..127;    --range extends to adjacent
                                     --words
     rank AT 4*word RANGE 0..31;
     serialnumber AT 5*word RANGE 1..31;
END RECORD;
```

where "word" is a constant equal to the number of storage units per target word. A variable-length-entry specified table will yield the alignment clause, "AT MOD 1* word". Ordinary tables with medium or dense packing will be translated using the locations of each component selected to conform to J73 semantics of the packing specifiers used. Tight tables will be effected by use of the pragma, "pack".

The preceding discussion has described the translation of serial tables to arrays of records. A parallel table will be translated to a record of arrays. The type of each of these arrays will be a record that is previously declared to include table item declarations grouped according to entry word. The general format of this translation is given as follows: a parallel table declaration

```
TABLE tt (44) PARALLEL...
BEGIN
  "declarations of items positioned in word 0"
  "declarations of items positioned in word 1"
  .
  .
  .
END
```

will be translated to the following declarations:

```
TYPE tt_word_0_type IS
RECORD
   --declarations of objects positioned in word 0
END RECORD;

TYPE tt_word_1_type IS
RECORD
   --declarations of objects positioned in word 1
END RECORD;
   .
   .
   .
TYPE tt_type IS   --a "record-of-arrays" type
RECORD
  tt_word_0: ARRAY(0..44) OF tt_word_0_type;
  tt_word_1: ARRAY(0..44) OF tt_word_1_type;
     .
     .
     .
END RECORD;

  tt:tt_type;    --declares a record object
```

Grouping the objects of each entry word in a separate record permits translation of parallel tables with specified entries using representation specifications for each record, including the positioning of several items per entry word. An ordinary table with parallel structure will not require these separate record type declarations for each entry word; it is completely described by a single record. For example, the table

```
TABLE ordinary (44) PARALLEL;
BEGIN
     ITEM aa A 0,31;
     ITEM bb S ;
     ITEM cc C 4;
END
```

will become

```
TYPE ordinary_type IS
RECORD
     aa:ARRAY (0..44) OF A0_31_type;
     bb:ARRAY (0..44) OF S_type;
     cc:ARRAY (0..44) OF C4_type;
END RECORD;

ordinary: ordinary_type;
```

Table presets and table item presets will be translated using aggregate values as described for string presets in 3.2.2.2.1. The "like" option will result in records which include reference to previously declared records as appropriate. Star-bound tables will be declared using unconstrained arrays ("ARRAY(<>)").

There is a special case in which specified table declarations will not be completely translated. J73 table items may overlap in bit position within a table entry. This programming technique is sometimes used to define mask fields and substrings of table data. Under the translation outlined in this paragraph, overlapping table items would be translated to incorrect Ada code, since locations specified by a representation specification within a record must not overlap. The exception to this rule is that storage for distinct variants of a record may overlap. However, this requires that the discriminants be static, prohibiting dynamic selection of variant objects. Thus, variant records will not be used, nor does any other Ada construct appear adequate for this mapping. The Translator will detect table item overlaps, translate them as (illegally) specified records, and output a warning message to inform the user of the need to reprogram.

3.2.2.2.3 Pointers

J73 pointer types will be translated to Ada access types. The translation is quite simple for typed pointers.

```
TYPE link P cell;
```

becomes

```
TYPE link IS ACCESS cell;
```

and

```
ITEM symptr P symtab;
```

is translated to the pair of declarations,

```
TYPE symptr_type IS ACCESS symtab;
symptr: symptr_type;
```

This permits an access of a pointed-to variable such as "variable@symptr" to be translated to "symptr.variable".

Translation of untyped pointers is more difficult, because Ada does not permit anonymous access types. The Translator must perform a global analysis of the program to determine the types of all objects to which the pointer may point. If the pointer is used for objects of only one type, the Translator will simply "type" the pointer in its declaration. For example, a table containing an untyped pointer

```
TABLE cell (49);
BEGIN
  ITEM value val_type;
  ITEM next P; "next is used to point to other cells"
END
```

will be translated to

```
TYPE cell_type; --incomplete type declaration
TYPE next_type IS ACCESS cell_type;
TYPE cell_type IS
RECORD
  value: val_type;
  next: next_type;
END RECORD;
```

```
    cell: ARRAY(0..49) OF cell_type;
```

If the pointer is used for objects of several types, the Translator will select a type for the pointer according to frequency of use. For example, if an item declared as an untyped pointer is most often used to point to objects of type "cell2_type", then

```
    ITEM pointanywhere P; "usually points to cell2_type"
```

will be translated to

```
    TYPE pointanywhere_type IS ACCESS cell2_type;
    pointanywhere: pointanywhere_type;
```

with an incomplete declaration of cell2_type included (if necessary) before the declaration of pointanywhere_type. References to pointanywhere will need type conversions only if the target type is not cell2_type; conversions to cell2_type will be deleted by the Translator, since they are unnecessary.

3.2.2.2.4 Other Declarations

Block declarations are used to declare groups of items, tables, and other blocks which are to be stored contiguously. Although no Ada construct provides contiguous storage allocation, blocks will be translated to records, providing access to blocks (including parameter passing) in a manner which is semantically similar to J73. In general, a block declaration of the form

```
BLOCK datagroup;
BEGIN
  "sequence of declarations"
END
```

will be translated to

```
TYPE datagroup_type IS
RECORD
 --sequence of declarations
END RECORD;
datagroup: datagroup_type;
```

along with a warning message to inform the user that the objects declared in the block/record may not have contiguous storage allocation.

Statement name declarations are used to declare labels which are to be used as formal parameters. Since the Translator will not translate label parameters, these declarations will not be translated (see 3.2.3.3).

Define declarations are used to achieve parameterized compile-time string substitution (i.e., macro-expansion). Define declarations which correspond to simple constants will be translated to constant declarations. For example,

```
DEFINE upperbound "2**15-1";
```

will be translated to

```
upperbound: CONSTANT:= 2**15-1;
```

Other define declarations, in general, have no Ada equivalent. The Translator will simply expand define calls in the J73 module before translation. The user may request a summary of define expansions performed as a translation option.

Although Ada contains no construct for overlaying data, an Ada implementation may provide a pragma for this purpose. The overlay declaration will be translated using this pragma if it is available; otherwise, overlay declarations will not be translated.

J73 allows null declarations whose syntactic form is either a semicolon or an empty BEGIN-END bracket. These declarations will be translated to the Ada construct, NULL.

3.2.3 Executable Constructs

This paragraph describes the translation functions associated with formulas, expressions, and statements. The discussion is given in three parts:

a. Expressions, formulas, and assignment statements.
b. Local control statements.
c. Procedure and function call statements and return statements.

Special executable constructs known as intrinsic functions are discussed in 3.2.5.

3.2.3.1 Expressions and Assignments

In general, arithmetic formulas such as ((AA*BB-CC)**2) will be unchanged by the Translator. Each arithmetic operator of J73 has an Ada equivalent with the same form and precedence. Ada distinguishes between binary and unary uses of the operators "+" and "-", giving higher precedence to unary occurrences, but in practice this does not affect the results of an arithmetic formula. (J73 treats unary "+" and "-" as "signs" rather than operators, so that expressions such as (5--3) must be written as (5-(-3)), removing the need for a precedence distinction.) Type qualifiers will be inserted into fixed point expressions when needed, as discussed in 3.2.2.1.

Status, table, character, and pointer formulas do not involve operators, and will not be changed by the Translator. Bit formulas in J73 are required to include parentheses whenever more than one kind of operator is used, so that precedence is irrelevant. The EQV operator will be translated to "=", which is overloaded in Ada to include boolean expressions. The AND and OR operators, when used in boolean formulas (bit formulas of type B1), will be translated to the short circuit forms, AND THEN and OR ELSE, corresponding to the J73 semantics for boolean formulas; bit formulas of size greater than one will use the standard AND and OR forms.

Relational operators are equivalent in J73 and Ada. The "not equal" operator in J73 ("<>") will be converted to its Ada equivalent, "/=". All relational operators have equal precedence in both languages.

Type conversions in Ada are permitted only between closely related types, so that conversions of numeric types to numeric types, bit types to bit types, character types to character types, and table types to table types may be translated directly. For example,

```
integer(xx) "xx is a halfword integer"
5,26 *)yy "yy is type A 0,31"
```

translated to

```
integer (xx)
26_type(yy)
```

ions between unrelated types (such as character to integer) and ions involving pointers, status objects, and the REP function be performed directly in Ada. The only Ada construct available ich conversions is the predefined generic function, ED_CONVERSION. Instantiations of this generic will appear in lefined_package for each kind of conversion which has no direct ivalent. The J73 conversions

```
8*)name    "name is of type C 1"
xyz)       "xyz is of type F"
le2(@point) "point is of type P table1"
```

cause the following instantiations to be included in lefined_package:

```
CTION C1_type_conversion IS NEW UNCHECKED_CONVERSION (C1_type);
CTION F_type_conversion IS NEW UNCHECKED_CONVERSION (F_type);
CTION table1_type_conversion IS NEW UNCHECKED_CONVERSION
le1_type);
```

the type conversions may be translated to the function calls

```
ype_conversion(name)
pe_conversion(xyz)
e1_type_conversion(point.all)
```

anslation technique will work correctly only if the Ada tation being used permits the unchecked conversions generated Translator. In J73, conversions betw en unrelated types are ned by compile-time rules, while Ada does not specify what ill be used by a compiler in performing (or rejecting) such ons. For any unchecked conversion which is not allowed by the mpiler, the user must replace the instantiation of D_CONVERSION with a customized function that emulates the nding J73 rules for the conversion.

Assignment statements will be translated by replacing the "=" with its Ada equivalent, ":=". Assignments to more than one variable in a single statement, such as

```
var1,var2,var3 = var2 + 6;
```

will be broken into separate assignments,

```
temp:= var2 + 6;
var1:= temp;
var2:= temp;
var3:= temp;
```

using a temporary variable to conform to the J73 rule that the right hand side be evaluated only once.

To a small extent, J73 programs may rely on the side effects of the order of evaluation of expressions and assignments. The language guarantees that the right-hand side of an assignment statement will be evaluated before the left-hand side, and that function arguments and table indices will be evaluated left-to-right before any expressions or assignments are performed. Dependence on side effects of these evaluations, while generally considered poor programming practice, is possible in J73. However, Ada gives no such guarantees regarding order of evaluation; a program which contains such side effect dependencies may be translated to an erroneous Ada program. The user is responsible for detecting and removing these dependencies.

3.2.3.2 Local Control Statements

This paragraph describes the translation of statements which affect a program's flow of control on a local basis. Global control constructs (call and return) are discussed in the following paragraph.

The syntax of J73 loop statements is relatively complex. A loop statement may contain, in addition to a loop parameter and a while clause, a by-phrase, a then-phrase, and an initial value. Futhermore, the loop parameter may be either a global program variable or an implicitly declared object which is local to the loop and unaccessible outside of the loop. By comparison, Ada loops are quite simple. They may contain an implicitly declared loop parameter, a discrete range for the parameter, and a while-clause; global loop parameters and explicit by- or then-clauses are not permitted. Translation of loop statements is a rare instance of mapping a complex J73 structure onto a simpler Ada structure.

Loop statements with no loop parameter are easily translated. In general, a loop of the form

```
    WHILE booleanformula;
    BEGIN
       .
       .
       .
    END
```

will be translated to

```
    WHILE booleanformula
    LOOP
       .
       .
       .
    END LOOP;
```

A loop with an implicitly declared loop parameter (a "control-letter") will be translated using an iteration clause (FOR loop_parameter IN range) whenever the by-clause or then-clause corresponds to a loop parameter increment of +1 or -1. For example,

```
    FOR i:0 BY 1 WHILE i<100;
     .
     .
     .
```

becomes

```
    FOR i IN 0..99
    LOOP
     .
     .
     .
    END LOOP;
```

and

```
    FOR i:22 THEN (i-1) WHILE i>=0;
     .
     .
     .
```

becomes

```
    FOR i IN REVERSE 0..22
    LOOP
     .
     .
     .
    END LOOP;
```

A loop with a control letter but no by-clause, then-clause, or while-clause can be translated without an iteration clause:

```
FOR i:1;
BEGIN
  .
  .
  .
END
```

will be translated to an Ada block with a declarative part

```
DECLARE               --block for loop statement
  i:INTEGER:=1;
BEGIN
LOOP
  .
  .
  .
END LOOP;
END;                  --block for loop statement
```

ensuring that the loop parameter is local to the loop statement. A similar loop with a global variable rather than a control letter, such as

```
FOR eventcount: v(firstevent);
BEGIN
  .
  .
  .
END
```

will be translated, without a block or declarative part, to

```
eventcount:=firstevent;
LOOP
  .
  .
  .
END LOOP;
```

since the loop parameter is already declared global to the loop statement.

Loop statements with global variable loop parameters and by-clauses, then-clauses, or while-clauses, as well as loops with control letters and increments not equal to +1 or -1, will be translated to Ada structures consisting of assignment statements and while-loops. For example, the loop

```
FOR aa:bb BY cc WHILE dd>ee; "aa,bb,cc,dd,ee are global"
BEGIN
  .
  .
  .
END
```

will be translated to

```
aa:=bb;
WHILE dd>ee
LOOP
  .
  .
  .
aa:=aa+cc;
END LOOP;
```

which is not only semantically identical to the J73 form, but should also run just as efficiently. Another example:

```
FOR i:bb BY cc WHILE i<>0; "i is a control letter"
BEGIN
  .
  .
  .
END
```

is translated to a block with a local declaration of i:

```
DECLARE   --block for loop statement
  i: INTEGER:=bb;
BEGIN
WHILE i/=0
LOOP
  .
  .
  .
i:=i+cc;
END LOOP;
END;      --block for loop statement
```

The exit statement in J73 is directly mappable to Ada. In general, exit statements will be unchanged by the Translator. A construct such as

```
WHILE condition1;
BEGIN
  .
  .
  .
IF condition2;
    EXIT;
  .
  .
  .
END
```

may be translated to

```
WHILE condition1
LOOP
  .
  .
  .
IF condition2 THEN
    EXIT;
END IF;
  .
  .
  .
END LOOP;
```

or, more cleanly,

```
WHILE condition1
LOOP
  .
  .
  .
 EXIT WHEN condition2;
  .
  .
  .
END LOOP;
```

Selection of the latter translation technique is an optimization that may be requested by the user as an option.

J73 IF statements translate straightforwardly, differing from Ada IF statements in that the reserved word "THEN" must precede the body of the statement. Therefore,

```
IF condition;
     "any statement"
```

is translated to

```
IF condition THEN
     --sequence of statements
END IF;
```

A complex IF statement such as

```
IF condition1;
     BEGIN
      .
      .
      .
     END
ELSE IF condition2;
          .
          .
          .
        ELSE
          .
          .
          .
```

will be translated using the ELSIF construct to

```
IF condition1 THEN
 .
 .
 .
ELSIF condition2 THEN
   .
   .
   .
ELSE
   .
   .
   .
END IF;
```

Case statements are also quite easily translated, with the construct "(case-index,...):" replaced by "WHEN case-index|...=>". For example,

```
    CASE expression;
    BEGIN
         (0,1):                   "statement1"
         (2:4,8):                 "statement2"
         (5):                     "statement3"
         (DEFAULT):               "statement4"
         (9,11):                  "statement5"
    END
```

is translated to

```
    CASE expression IS
         WHEN 0..1=>              --statement1
         WHEN 2..4|8=>            --statement2
         WHEN 5=>                 --statement3
         WHEN 9|11=>              --statement5
         WHEN OTHERS =>           --statement4
    END CASE;
```

The default case alternative is moved to the end of the statement, as is required in Ada. The FALLTHRU construct, which causes case alternatives to be executed sequentially, has no Ada equivalent; each appearance of FALLTHRU will cause the statements of the following case alternative to be duplicated at the end of the case alternative which contained the FALLTHRU.

The final statement discussed in this paragraph, the GOTO statement, will be unchanged by the translator. The resulting Ada program will be correct as long as none of the GOTO's cause a transfer of control into an if statement or a case statement. J73 permits such transfers, while Ada does not. The Translator will detect and flag such GOTO's, informing the user of the need to restructure the module.

3.2.3.3 Call and Return Constructs

Procedure and function calls in J73 are syntactically similar to their Ada equivalents. Parameter passing mechanisms are semantically different: J73 specifies the way an argument will be passed to and used by a subroutine, while Ada specifies only the effect a subroutine may have on an argument. The difference between these two approaches involves the copying of actual parameter values.

J73 semantics for value binding and result binding require that a copy of the parameter is used by the subroutine. Ada provides two parameter modes, IN and IN OUT, which require copies of scalar and access type arguments, but not of composite (record or array type) arguments. To ensure that composite arguments are passed by copying, the Translator must generate explicit assignment statements to copy composite parameters into and out of temporary locations whenever value or result binding is used for blocks and tables.

In general, J73 input parameters will be translated to Ada IN parameters, and J73 output parameters will be translated to Ada IN OUT parameters. For example, the procedure declarations

```
    PROC swap (:aa,bb); "aa and bb are integer output arg's"

    PROC update (newvalue: buffer); "newvalue is floating,
                                     buffer is a table"

    PROC tablecopy (BYVAL table1); "table1 is an input value arg"
```

will be translated to

```
    PROCEDURE swap (aa,bb:IN OUT integer); --value result binding

    PROCEDURE update (newvalue: IN F_type; buffer: IN OUT
         buffer_type);
         --value binding for newvalue, reference binding for buffer

    PROCEDURE tablecopy (table1: IN table1_type);
         --value binding
           .
           .
           .
      table1_temp:=table1;
              --ensures that a copy of the argument is used
           .
           .
           .
             --references to table1_temp rather than table1
```

Explicit copying of value or result bound composite parameters, as in the third example, may be suppressed by the user if desired. Arguments of functions will be translated the same way as procedure arguments. Reference binding, which is used in J73 by default for tables and blocks, will be translated to IN or IN OUT parameter binding in the hope that the Ada implementation to be used will use a reference mechanism for such parameters. If the implementation uses a copying mechanism, then the subroutine may have an undesired effect if its context is changed during a run-time interrupt. However, it would appear unlikely that an implementation would ever use a copying mechanism for composite parameters, since reference mechanisms are generally much more efficient.

Subroutine name parameters will be translated to enumeration objects. For example, the procedure

```
DEF PROC p1 (tobecalled);
     .
     .
     .
     PROC to be called;
       .
       .
       .
```

in which the formal parameter "tobecalled" may assume the actual values "p5", "p6", or "p7," will become the package

```
WITH p5_package, p6_package, p7_package;
PACKAGE p1_package IS
     TYPE tobecalled_type IS (p5, p6, p7);
     PROCEDURE p1 (tobecalled: IN tobecalled_type);
     .
     .
     .
```

Using this translation, a call to the formal parameter is translated to a case statement:

```
tobecalled; "call to the procedure associated with
             the formal parameter"
```

becomes

```
CASE tobecalled IS -- which proc to call?
     WHEN p5 => p5; --call p5
     WHEN p6 => p6; --call p6
     WHEN p7 => p7; --call p7
END CASE;
```

in which the procedure names "p5" "p6" and "p7" are overloaded by enumeration literals with identically spelled names. Thus, the construct

```
WHEN p5 => p5;
```

means, "when the value of the parameter "tobecalled" is the enumeration literal "p5", call the procedure named "p5" (declared in p5_pacakge)." The overloading of the procedure names will be unambiguously resolved by the Ada compiler.

ABORT phrases and ABORT statements are similar to Ada exception handlers and RAISE statements in that they result in termination of a subroutine without binding the values of output parameters. However, there is a crucial difference between ABORT's and RAISE's: Raising an exception causes control to be transferred to a special section of code (an exception handler) at the end of a block or procedure body, and may not transfer control (GOTO) back into any other place in the block or procedure. In contrast to this well-structured method of prematurely terminating a procedure in Ada, the J73 ABORT causes a virtually unrestricted GOTO (to any part of a calling procedure) which cannot be effected using an exception mechanism. Therefore, ABORT phrases and statements will not be processed by the Translator. The user may restructure the calling routine so that it can use an exception mechanism; usually, this will not be difficult to do by hand. Similarly, statement name parameters and GOTO statements with formal statement name parameter targets, which are special cases of the ABORT mechanism, will not be automatically translated.

Procedure calls and function calls will be translated using positional syntax, as in J73, so that calls will be unchanged by the Translator. The only exception is that calls to parameterless functions, such as

```
currenttime = systemclock; "call to function with no arg's"
```

will use empty parentheses,

```
currenttime:=systemclock();
```

as is required in Ada. Return statements in procedures will be unchanged by the Translator, consisting simply of the reserved word RETURN. Functions will use the following translation technique:

a. assignments to the function name will be translated to assignments to a dummy variable.

b. "RETURN" will be translated to "RETURN dummy_variable".

For example, the function

```
PROC cuberoot (:number) A 10,21;  "number is type A 13,18"
BEGIN
   .
   .
   .
   cuberoot = expression;
   RETURN;
END
```

will be translated to

```
    FUNCTION cuberoot (number: IN OUT A13_8_type_) RETURN
         A10_21_type IS

         cuberoot_result: A10_21_type;
         BEGIN
            .
            .
            .
            cuberoot_result:=expression;
            RETURN cuberoot_result;
         END cuberoot;
```

This technique will be used to translate each function-name assignment and each return statement occurring within a function. Procedures and functions declared as INLINE will result in the insertion of the pragma, "INLINE procedure_name", into the program at the point of declaration.

The two remaining types of J73 statements, stop statements and null statements, are translated as RAISE system_stop; and NULL; respectively. The former statement will raise an exception called "system_stop" which is user supplied (or may be supplied by an implementation). If an integer formula is included, such as

```
    STOP 22;
```

the Translator will generate an assignment to the variable system_stop_value before raising the exception:

```
    system_stop_value:=22;
    RAISE system_stop;
```

The semantics of the value associated with the stop statement are implementation dependent in both languages. Declarations of this exception and variable will be included in J73_predefined_package.

3.2.4 Directives

J73 provides 22 directives. Ten of these directives will be translated; the others have no Ada equivalents.

The compool directive (!COMPOOL) is translated as described in 3.2.1.1. The copy directive (!COPY) will be translated to PRAGMA INCLUDE, having the identical effect of incorporating an external file into the program text at the textual location of the directive. The skip directive (!SKIP), along with its delimiters (!BEGIN and !END), will cause the Translator to insert comment delimiters ("--") before each line of text in the J73 module between the begin and end directives. The translated module will then include the non-translated J73 code as comments, along with a message informing the user of the presence of the skip directive.

The linkage directive (!LINKAGE) will be translated to the interface pragma, "PRAGMA INTERFACE (language_name, subprogram_name)", where language_name is provided by a TPF entry and subprogram_name is the name of the procedure or function which used the linkage directive. The listing directives, !LIST and !NOLIST, will be translated to "PRAGMA LIST (ON)" and "PRAGMA LIST (OFF)"; the eject directive, !EJECT, will be translated to the form feed symbol used by the Translator's host environment (unless the Ada implementation to be used features an eject pragma, in which case that pragma will be used). The initialize directive, !INITIALIZE, has no Ada equivalent, but will be effected by generating a preset of zeroes for all static data declared in the scope of the directive. That is, ":=0" or ":=0.0" or "(0..99=>0.0)", etc., will be inserted into the declaration of each static object.

Nine of the J73 directives (!TRACE, !INTERFERENCE, !REDUCIBLE, !BASE, !DROP, !ISBASE, !LEFTRIGHT, !REARRANGE, and !ORDER) have no predefined Ada equivalent. However, a particular Ada environment will probably include features which are identical (or at least similar to) many of these directives. The Translator will use any such features which are available via TPF entries for each directive. In the absence of a TPF entry, the directive will not be translated.

The remaining directives, !LISTINV, !LISTEXP, and !LISTBOTH, will be discarded by the Translator; define substitutions are not translated per se (see 3.2.2.2.4), so that these directives are not meaningful.

3.2.5 Intrinsic Functions

Most J73 intrinsic functions have Ada equivalents. Translation of these intrinsics is summarized as follows:

J73	Ada
LOC(table1)	table1'ADDRESS
BIT(mask1,5,8)	mask1(5..12)
BYTE(name,0,1)	name(0..0)
ABS(climbrate)	ABS(climbrate)
BITSIZE(table1)	table1_type'SIZE
BYTESIZE(name)	name_type'SIZE/BITSINBYTE
WORDSIZE(mask1)	mask1_type'SIZE/BITSINWORD
LBOUND(table1,2)	table1_type'FIRST(2)
UBOUND(table1)	table1_type'LAST
NWDSEN(table1)	table1_type'SIZE/BITSINWORD
FIRST(points)	points_type'FIRST
LAST(color)	color_type'LAST

In the examples above, the BIT and BYTE functions are translated to slice notation as discussed in 3.2.2. Many of the intrinsic functions involving object size or position are translated to predefined attributes of the objects' types.

The remaining J73 intrinsics, NEXT, SHIFT, and SGN, will be translated to syntactically equivalent calls to predefined functions (except for NEXT(status_type_variable), which translates directly to enumeration_type'SUCC(variable)).

The function NEXT will be declared in J73_predefined_package as

```
GENERIC
     TYPE enum IS (<>);
FUNCTION next (name:enum;number:integer) RETURN enum IS
BEGIN
     IF (number>0) THEN
        FOR i IN 1..number LOOP
           name := enum'SUCC(name);
        END LOOP;
     ELSE  -- number is =<0
        number := -number;
        FOR i IN 1..number LOOP
           name := enum'PRED(name);
        END LOOP;
        RETURN name;
END next;
```

a function call such as

T(color, 2) "second successor of color"

translated to a generic function call

T(color,2) --same as J73 version

uding the instantiation

ICTION next_color IS NEW next(color_type);

translation of the module. A similar generic must be supplied user to overload NEXT for access types (in an implementation nt manner) if the NEXT function is used on pointers. The SHIFT GN functions will be provided by the Translator in defined_package as generics similar to NEXT, so that expression as SHIFTR(xx,5) and SGN(aa) can be translated using iations such as

ICTION shiftr_xx IS NEW shiftr (xx_type);

ICTION sgn_aa IS NEW sgn (aa_type);

Miscellaneous Functions

aragraph includes a discussion of several issues which have not xplicitly covered by previous paragraphs, including translation es and comments, output listing format of the translated Ada and translation warning messages.

Names

which are not Ada reserved words and which do not contain the characters "'" or "$" will be unchanged by the Translator. The er "'" will have a default translation of "_"; "$", appearing first character of a name, will be translated to "S_"; a "$" 1 in a name will have a default translation of "_S_". Names re identical to Ada reserved words will be changed to include sion "_user". Labels will be delimited by <<...>>, as required Some examples of name translation are given below:

J73 name	Ada name
airspeed	airspeed
dot'product	dot_product
$status	S_status
main$cycle	main_S_cycle
loop	loop_user
statementlabel	<<statementlabel>>

Names of status constants will be translated by removing the "v" and the parentheses, so that "v(red)" becomes simply "red". The status constant name will also be subject to the rules described above for special characters and reserved words, and will be qualified with a type name if necessary, as discussed in 3.2.2.2.1.

Each of the rules for name translation (except label bracketing) may be overridden by the user, if desired. This is accomplished with TPF entries for each rule. For example, the user may wish to translate the "$" to "_x_", reserved words such as "loop" to "user_loop", or a status constant to "v_red". The user may prefer "proc1_package" to "proc1_pack" or "table1_record" to "table1_type". These preferences can be indicated with TPF entries.

The Translator is responsible for detecting name conflicts for all names, whether user generated or Translator-generated. For example, if the module being translated contains the names "range" and "range'user", a conflict will occur: both names will be translated to "range_user". The Translator must inform the user of the need to change either one of the names' spelling or to modify the TPF entry for one of the two cases (translation of "'" or translation of Ada reserved words).

J73 implementations normally permit lower case letters to be used (the basic character set is upper case). Ada also uses upper case as its basic character set, and will presumably allow lower case in most implementations. In both languages, corresponding upper and lower case characters are considered equivalent (except in character literals, where they are distinct). The Translator will use both cases, as in the examples of code given throughout this document, unless the user wishes otherwise. A TPF entry is provided for this purpose.

3.2.6.2 Comments

Although comments have no semantic effect on a program, the Translator will attempt to preserve all comments appearing in the module being translated. In most cases, a single J73 statement with a comment will translate to a single Ada statement with a comment. Source lines consisting of nothing but comments are also easily handled:

```
"This is a comment
 that uses three lines
 of the program"
```

becomes

```
--this is a comment
--that uses three lines
--of the program
```

However, comments may be embedded within a statement or declaration, such as

```
IF (aa<18.5) "below threshold" AND (bb>0);
```

Since Ada comments always extend to the end of the line, an embedded comment will either be moved to the end of the line

```
IF (aa<18.5) AND (bb>0);    --below threshold
```

or will be left in place while the remainder of the statement is moved to the next line:

```
IF (aa<18.5)    --below threshold
     AND (bb>0);
```

Selection of which technique is used is left as a user option. Another problem occurs when a single J73 statement or declaration is translated to more than one Ada statement or declaration. An example given in 3.2.3.2, for example, maps a for statement into two assignment statements and a while statement. In this case, the comment will be placed with the "key" statement of the Ada translation:

```
FOR aa:bb BY cc WHILE dd>33; "loop through all entries"
```

will be translated to

```
aa:=bb;
WHILE dd>ee --loop through all entries
LOOP
   .
   .
   .
aa:=aa+bb;
END LOOP;
```

The Translator will also create comments for Ada code which is Translator-generated. For example,

```
    TABLE employees (99); "personnel records"
    BEGIN
       .
       .
       .
    END
```

will result in

```
    TYPE employees_type IS
    RECORD          --describes body of table "employees"
       .
       .
       .
    END RECORD;
    employees: ARRAY (0..99) OF employee_type;     --personnel records
```

The Translator will also generate comments to inform the user of the purpose of a with clause:

```
    WITH comp1;         --includes items aa,bb
                        --and tables tab1, tab2
```

The user may optionally suppress either original comments or Translator-generated comments.

3.2.6.3 Prettyprinting

The Ada modules output by the Translator will be printed in a format which corresponds to commonly accepted style for high order language programming. Statements within logical blocks such as procedures, loops, and records will be indented one tab stop (three spaces) relative to the enclosing block. Single spaces will be insert_d between names, operators, and reserved words. The user may select either upper or lower case letters to be used for either reserved words or names. In general, the code will be formatted like the examples given throughout this Functional Description.

Warning messages will be inserted into the output text as necessary. The messages will correspond to three levels of severity. Level 1 warnings inform the user that the Translator has made some assumption (presumably a valid assumption) about the programming environment. For example, when translating an assignment to a floating point type variable which was declared with a rounding option, the message

```
    --**L1warning: assumes presence of a rounding procedure**
```

will be inserted as the line following the call to the rounding procedure. Level 1 warnings are informational and may be suppressed by the user if desired.

A translation which introduces a possible syntactic or semantic error will be accompanied by a Level 2 warning message. Examples:

```
--**L2warning: record component overlap is illegal**
--**L2warning: target of goto is inside a compound statement**
```

Untranslated constructs will be flagged by Level 3 warning messages, such as

```
--**L3warning: define declaration not translated**
--**L3warning: order directive not translated**
```

Warning messages are printed as Ada comments so that the module may be compiled, if desired, without modification.

3.3 Inputs-Outputs

This paragraph describes the input and output requirements of the Translator.

3.3.1 Input Data

Three kinds of data are required as input to the Translator: user commands, J73 source, and translation parameters.

3.3.1.1 User Command Input

Users of the Translator must provide whatever host-dependent commands are required to invoke the Translator and specify input and output data file names.

3.3.1.2 J73 Source Input

The J73 source to be input for a single run of the Translator may be any portion of the program that is separately compilable by a J73 compiler, ranging from a single compool, procedure, or function to the entire program. All J73 code must be syntactically correct, which implies that it has been previously checked by either a compiler or a code auditor. Previous compilation or auditing of the J73 source code is not manditory; however, because the Translator will not perform syntax checking, reliable translation will result only from input that is absolutely free of syntax errors. Input which is syntatically correct but erroneous will, in general, have unpredictable results. Some specific instances of erroneous program translation have been discussed in previous sections.

3.3.1.3 Translation Parameter File

The Translation Parameter File (TPF) will be used by the Translator to guide the translation of J73 constructs whose mapping to Ada is either arbitrary or indefinite. Examples of such cases are variable names containing the "$" or "'" characters, variable and constant names which may be optionally qualified with a package or type name, optional insertion of constraint specifications and exception handlers, and selection of subroutine argument-passing modes. The TPF will be user accessible and may optionally be included as part of the Translator's output listing (along with the Ada program itself). Certain TPF entries may be overridden by user command inputs so that a single module can be translated in a special manner without modifying the TPF.

3.3.2 Output Produced

The Translator will produce three kinds of output: translated Ada modules, generated Ada modules, and a program dictionary.

3.3.2.1 Translated Ada Module Output

The major output of the Translator will be a listing of the Ada module produced by a run of the Translator. This listing will be appropriately formatted ("prettyprinted") to conform to standard programming practices, including indentation to exhibit nesting, alignment of "begins" and "ends", and form feeds for modular units (i.e., a new procedure gets a new page). Comments from the input J73 program will be included in the Ada listing if requested by the user. Warning messages will clearly delimit any missing Ada code corresponding to untranslated J73. The listing may be output to either a hard copy device (printer) for human inspection or to a file (disk or tape) for storage.

3.3.2.2 Generated Ada Module Output

Predefined types and intrinsic functions in J73 which have no exact Ada equivalent will require the generation of special modules. These modules will be Ada packages which specify predefined types unique to J73, as well as packages which either implement or at least specify J73 intrinsic functions. In the latter case, intrinsics whose implementation is target dependent rather than language dependent will be represented by a package specification with a body stub. This will permit the user to implement the function at a later date while ensuring syntactically correct references to the function immediately. The use of the "generated packages" will render the translated Ada program readable, since the resulting Ada syntax will be identical to the original J73 syntax for predefined/intrinsic constructs. In addition to clarity, efficiency and flexibility will be maintained: the packages generated by the Translator may be changed or replaced by the user with no syntactic impact on any of the translated modules.

3.3.2.3 Program Dictionary Output

For translation purposes, the Translator must keep an internal dictionary of the names of all modules and externals used in the program being translated. A listing of this dictionary may be output upon request of the user. It will contain the name of each library unit in the Ada translation, as well as external names listed according to which library unit contains either a definition of or a reference to each external.

3.4 Data Characteristics

The storage and characteristics of the data elements used by the Translator are summarized in the table below.

File Description	Mode	Format	Recommended Device Type
J73 Source	input	character	sequential or direct access
Translation Parameter File (TPF)	input	character	direct access
List of J73 Modules by File Name	input	character	direct access
Workspace	internal	binary	direct access
Dictionary	output	character	hard copy
Ada Modules	output	character	sequential, direct access or hard copy

The sizes of these elements are entirely dependent on the size of the J73 source program being translated (except for the TFF, which will require a fixed storage size of about 1K words).

3.5 Failure Contingencies

No failure contingencies are required for this system.

APPENDIX 1

SUMMARY OF PROBLEMATICAL CONSTRUCTS

Construct	Problem	Discussed in Paragraph
Specified tables with overlapping items	Illegal in Ada.	3.2.2.2.2
Contiguous storage allocation (Blocks) and overlays	Contiguous storage is not guaranteed; over-layed storage may not be provided in an Ada implementation.	3.2.2.2.4
Statement name declarations	No similar Ada construct.	3.2.2.2.4
Define declarations	Define's are expanded rather than translated.	3.2.2.2.4
Expressions with side effects	Side effects are not guaranteed.	3.2.3.1
Label parameters and abort statements	No similar Ada construct.	3.2.3.3
Directives	Certain directives may not be provided in an Ada implementation.	3.2.4

APPENDIX 2

MIL-STD-1589B CROSS REFERENCE

This appendix provides a cross-reference for J73 constructs according to the sections of MIL-STD-1589B. For each section or group of related sections of 1589B, the subparagraph of this Functional Description which is applicable is given in the right column.

1589B Section		Discussed in Paragraph
1.1	Complete Program	3.2.1
1.2.1	Compool Modules	3.2.1.1
1.2.2	Procedure Modules	3.2.1.2.1
1.2.3	Main Program Module	3.2.1.2.1
1.2.4	Conditional Compilation	3.2.4
1.3	Scope of Names	3.2.1.2, 3.2.6.1
1.4	Implementation Parameters	3.2.2
2.0	Declarations	3.2.2
2.1	Data Declarations	3.2.2.2
2.1.1	Item Declarations	3.2.2.2.1
2.1.1-6	Type Descriptions	3.2.2.1, 3.2.2.2.1
2.1.7	Pointer Type Descriptions	3.2.2.2.3
2.1.2	Table Declarations	3.2.2.2.2
2.1.2.1-4	Table Dimensions, Structure, Entries	3.2.2.2.2
2.1.3	Constant Declarations	3.2.2.2.1
2.1.4	Block Declarations	3.2.2.2.4
2.1.5	Allocation of Data Objects	3.2.1.2
2.1.6	Initialization of Data Objects	3.2.2
2.2	Type Declarations	3.2.2.2
2.3	Statement Name Declarations	3.2.2.2.4
2.4	Define Declarations	3.2.2.2.4
2.4.1	Define Calls	3.2.2.2.4
2.5	External Declarations	3.2.1.2.2
2.6	Overlay Declarations	3.2.2.2.4
2.7	Null Declarations	3.2.2.2.4
3.0	Procedures and Functions	3.2.1.2.1
3.1	Procedures	3.2.1.2.1, 3.2.3.3
3.2	Functions	3.2.1.2.1, 3.2.3.3
3.3	Parameters	3.2.3.3
3.4	Inline Procedures and Functions	3.2.3.3
3.5	Machine Specific Procedures	3.2.1.2.1

Appendix 2 - MIL-STD-1589B Cross Reference - Continued

1589B Section		Discussed in Paragraph
4.0	Statements	3.2.3
4.1	Assignment Statements	3.2.3.1
4.2	Loop Statements	3.2.3.2
4.3	If Statements	3.2.3.2
4.4	Case Statements	3.2.3.2
4.5	Procedure Call Statements	3.2.3.2
4.6	Return Statements	3.2.3.3
4.7	Goto Statements	3.2.3.2
4.8	Exit Statements	3.2.3.2
4.9	Stop Statements	3.2.3.2
4.10	Abort Statements	3.2.3.1
5.0	Formulas	3.2.3.1
6.0	Data References	3.2.1, 3.2.2, 3.2.6.1
6.1	Variables	3.2.2.2
6.2	Named Constants	3.2.2.2.1
6.3	Function Calls	3.2.3.3
6.3.1-11	Intrinsic Functions	3.2.5
7.0	Type Matching and Conversions	3.2.2.1, 3.2.3.1
8.1	Characters	3.2.6.1
8.2	Symbols	3.2.6.1
8.3	Literals	3.2.2.1, 3.2.2.2.1
8.4	Comments	3.2.6.2
8.5	Blanks	3.2.6.3
9.0	Directives	3.2.4

SYSTEM/SUBSYSTEM SPECIFICATION

for the

JOVIAL (J73) TO ADA TRANSLATOR

Prepared by:

Mark J. Neiman

TABLE OF CONTENTS

LIST OF FIGURES

'ION 1. GENERAL

Purpose of the System/Subsystem Specification

System/Subsystem Specification for the JOVIAL (J73) to Ada ıslator Investigation (F30602-81-C-0127) is written to fulfill following objectives:

a. To provide definition of a proposed system to translate JOVIAL (J73) programs to Ada programs.

b. To communicate details of the on-going analysis between potential users and potential development personnel.

Project References

›rietary Software Systems is under contract to the Rome Air ›lopment Center to investigate the automatic translation of :AL (J73) to Ada. The system proposed in this document is ›nded to provide production quality translation of JOVIAL)) programs to Ada in accordance with the Functional :ription (10 January 1982) and the Statement of Work (PR No. ·3289) for the project. In addition to these documents, ›rences listed in Section 1.2 of the Functional Description also pertinent to the project and will be cited within this ıment.

Terms and Abbreviations

following terms and abbreviations will be used throughout System/Subsystem Specification:

IA — Descriptive Intermediate Attributed Notation for Ada.

neous — A high order language program which contains one or more violations of language semantics which are not detected by a compiler. Erroneous programs have unpredictable run-time results.

rnal — A program element that is referenced by modules which are compiled separately from the module in which the element is declared.

The programming language JOVIAL (J73) as specified by MIL-STD-1589B.

Module	A portion of a J73 or Ada program which is logically distinct from the rest of its program and which may be compiled or translated separately.
Parse Tree	A data structure which represents the abstract syntax of a high order language program or module.
Program	All of the modules of a J73 or Ada program, as opposed to an individual compilation unit.
TPF	Translation Parameter File - a user accessible file which specifies which translation options will be used for a run of the Translator.
Translator	The proposed JOVIAL (J73) to Ada translator.

SECTION 2. SUMMARY OF REQUIREMENTS

2.1 System/Subsystem Description

The Translator consists of a computer program and related data needed to automatically translate a J73 program to an equivalent Ada program. The primary inputs to the Translator are J73 source modules and the Translation Parameter File (TPF). The Translator produces two kinds of output listings: Ada source modules (with diagnostics) and a program dictionary.

The purpose of the Translator is to provide a high degree of automation to the process of converting a correct J73 program to an equivalent Ada program. The J73 program must be correct in the sense that it contains no syntactic or semantic errors (i.e., it is a "debugged" program). Figure 2-1 illustrates the use of the Translator in the software conversion process; Figure 2-2 shows the major functional components of the Translator itself.

```
                 BEGIN
                   |
                   v
                 .....
               . J73    .
              .  SOURCE   .
               .MODULES.
                 .....
                   |                            ---------------
                   |                            | J73 SOURCE  |
                   |             ---------------| CORRECTIONS |
                   |             |              ---------------
                   v             v                     ^
           -------------------------------             |
           |  INIT             TRAN      |             |
           |                             | Translation errors |
           |                             |--------------------
           |       TRANSLATOR            |
           |                             |
           -------------------------------
                   |
                   |
                   v
                 .....
               . ADA    .
              .  SOURCE   .Translation Errors
               .MODULES.----------------------
                 .....                       |
                   |                         v
                   |                  ---------------
                   |                  | Ada Source  |
                   |                  | Corrections |<---
                   |                  ---------------   |
                   |                         |          |
                   |--------------------------          |
                   v                                    |
            ----------------                            |
            | Ada Compiler |                            |
            ----------------                            |
                   |                                    |
                   v                                    |
           ------------------                           |
           | Test/Integrate |----------------------------
           ------------------
                   |
                   v
                  END
```

Figure 2-1: Software Conversion Cycle
with Automatic J73-to-Ada Translation

```
                     --------
                     | EXEC |
                     --------
                        |
            ---------------------------------------------
            |                       |                   |
        --------                --------            --------
        | INIT |                | TRAN |            | LIST |
        --------                --------            --------
            |                     |   |
            ----------------------    |
                      |               |
                 -----------          |
                 | ANALYZE |          |
                 -----------          |
                                      |
                                      |
                                      |
                                      |
                                      |
                     -------------------------------
                     |                |            |
                 --------         --------     --------
                 | SPEC |         | BODY |     | GEN  |
                 --------         --------     --------
```

EXEC	:	MAIN EXECUTIVE
INIT	:	GLOBAL ANALYSIS INITIALIZATION
TRAN	:	EXECUTIVE FOR MODULE TRANSLATION
ANALYZE	:	MODULE ANALYZER
SPEC	:	J73-TO-IL FOR PACKAGE SPECIFICATION
BODY	:	J73-TO-IL FOR PACKAGE BODY
GEN	:	GENERATE ADA FROM IL
LIST	:	OUTPUT LISTINGS

Figure 2-2: Translator Structural Components

2.2 System/Subsystem Functions

The functions of the Translator are summarized in this paragraph. A complete description of these functions, including details, examples, and exceptions, appears in the Functional Description.

In translating a J73 program to Ada, the Translator will preserve the modular structure of the program. This is accomplished by translating compools and procedures to packages. The J73 constructs which reference separately compiled modules, the REF and the compool directive, are translated to Ada WITH clauses. A J73 procedure "P1" containing compool directives, REF declarations, DEF declarations, and local declarations will become an Ada package with the general form:

```
WITH ...  -- names of other packages containing
     ...  -- compools and declarations of REF'ed objects
PACKAGE   P1_package IS
     PROCEDURE P1 ;    -- specification of P1
     ...               -- declarations of other DEF'ed objects
END P1_package;        -- end of package specification

PACKAGE BODY P1_package IS
     ...               -- declarations of local STATIC objects
     PROCEDURE P1 IS   -- body of P1
     BEGIN
     ...               -- remaining local declarations
     ...               -- rest of the procedure body
     END P1 ;          -- end of P1 body
END P1_package;        -- end of package body
```

This translation technique is valid for all compools and procedures except compools containing REF declarations and procedures containing partial compool inputs. The translations performed in these cases are discussed in the Functional Description.

The predefined types of J73 (signed and unsigned integer, fixed and floating point, character, and bit types) will be defined in a Translator-generated package called "J73_predefined_package." Declarations which use these types will be translated to Ada declarations which use similar type names (such as A14_1_type for A 14,1) and preserve all the attributes and type matching properties defined by J73 semantics. Bit and character types are implemented as arrays, so that the "slice" and "aggregate" notations are used to denote objects of bit or character type. Status types translate straightforwardly to enumeration types. Serial tables are translated to arrays of records, where each record is an entry of the table; parallel tables become individual records, in which each record component is an array.

Specified representation attributes of status types and table types will be achieved using Ada's representation specification constructs. Pointer types are translated to access types; block types will become record types. Declarations which are not perfectly translated include specified tables with overlapping item positions, statement name declarations, and overlay declarations. Define calls are expanded inline, so that define declarations are not translated per se.

Executable constructs are similar in J73 and Ada. Arithmetic and logical operations in the two languages have matching operators and precedences, so that translations will not require extra parentheses or special functions. Type conversions between closely related types are also straightforward, but conversions between unrelated types and conversions involving pointers, status objects, and the REP function are translated to calls to the generic function, UNCHECKED_CONVERSION. Assignment statements translate directly, with assignments to several variables in a single statement translated to several separate assignments. Side effects of expression evaluation order and assignment evaluation order will not be preserved by the Translator. Control statements (FOR, IF, and CASE) translate to the corresponding Ada statements, with major restructuring required on certain classes of FOR statements and minor restructuring of some CASE statements.

Procedures and function calls are translated to syntactically similar Ada calls, with input parameters passed in IN mode and output parameters passed in IN OUT mode. Code that explicitly copies value or result bound parameters is generated by the Translator for cases in which Ada does not guarantee the necessary value or result binding mechanisms. Label parameters, subroutine name parameters, and abort statements are not translated; they must be hand coded using, for example, an exception mechanism.

Ten of the 22 directives provided by J73 have simple Ada equivalents. The three directives related to define expansions are not needed; the remaining directives (!TRACE, !INTERFERENCE, !REDUCIBLE, !BASE, !DROP, !ISBASE, !LEFTRIGHT, !REARRANGE, and !ORDER) will be translated only if the Ada implementation to be used provides corresponding constructs, since no such constructs are predefined in the language.

Ada provides predefined attributes of types which are used for translation of most J73 intrinsic function calls. The BIT and BYTE functions are translated to slice notation. The NEXT, SHIFT, and SGN intrinsics will be translated to generic functions declared in J73_predefined_package.

The Translator will process names in a highly flexible manner. The user may control the translation of names containing the "$" and "'" characters, as well as the names of Translator-generated objects, using TPF entries. The Translator will detect any naming conflicts or violations; it will also preserve the original comments and create comments for Translator-generated code. Diagnostics will be embedded in the output listing to inform the user of assumptions or inaccuracies in the translation of a module. The output listing will conform to normal standards for structured programming with regard to format, indentation, etc.

2.2.1 Accuracy and Validity

The translations performed by the Translator will be accurate in the sense that the resulting Ada programs will be semantically equivalent to the J73 programs from which they were derived to the largest extent possible. Except for certain untranslated constructs, which will be clearly flagged in the output, the Ada produced by the Translator will be a valid Ada program in that

a. It will contain no syntax errors;

b. Any missing code that is required for execution of the program will be clearly identified;

c. It will be compilable in a standard Ada environment without modifications (such as reorganizing statements and declarations or renaming modules or variables);

d. It will conform to general standards for readable, well structured programming.

In general, two versions of a program cannot be guaranteed to have absolutely identical run-time behavior in two different environments, even if the versions were generated from the sam source code (e.g., a J73 program compiled for two different targets). Therefore, the Translator cannot be required to produce a "perfect" translation of a non-trivial program. However, it will be required to preserve the original program semantics wherever possible, at the expense of some run-time efficiency if necessary, and to inform the user of any possible deviations from J73 semantics that are introduced by the translation.

2.2.2 Timing

Although portions of a program may require repeated translation to resolve various translation problems, the overall translation process will be a one-time task. High performance with respect to throughput is, therefore, not given a high priority. The Translator should process J73 source code at about the same speed as a compiler, roughly 100 source lines per CPU minute on a fast mainframe host system.

2.2.3 Flexibility

Flexibility in the Translator is provided by use of the Translation Parameter File, which is discussed in Section 4.3.1.

SECTION 3. ENVIRONMENT

3.1 Equipment Environment

A general purpose, medium scale mainframe computer will be needed to support the Translator and its associated data. The host environment must include enough direct access memory to store the Translator, the J73 program being translated, and all related data, such as symbol tables, intermediate representations of the modules under translation, and output data. A host environment which is capable of supporting storage and compilation of a given J73 program will be adequate for support of the translation to Ada of that program; no new processors, memories, or input/output devices will be required.

3.2 Support Software Environment

The Translator will operate under control of a general purpose operating system. Invocation of the Translator, specification of input and output files, and modification of J73 code for re-translation (as shown in Figure 2-1) will require the job control, file management, and text editing capabilities which are provided by a typical operating system on a medium scale computer. No new support software should be necessary. The Translator could be integrated into an Ada Programming Support Environment (APSE), but this is not an inherent requirement. For example, if the Translator were implemented in Ada, an APSE would be necessary for maintenance and run-time support; however, if it were implemented in J73, a J73 compiler (and linker) would be needed -- an APSE would be unnecessary.

SECTION 4. DESIGN DETAILS

4.1 General Operating Procedures

To translate a J73 program to Ada, the user must successfully complete two sets of tasks. First, the translation process must be initialized; second, individual modules can be translated to Ada modules.

4.1.1 Initializing the Translator

The translation process is begun by invoking the Translator in INIT mode. The inputs required in this mode are the TPF, the program module list, and all of the source files of the J73 program to be translated (see Section 4.3.1 for detailed discussion of these inputs). When the Translator runs in INIT mode, it will use the TPF and the module list to perform a global analysis of the J73 program. The initialization process must be repeated if a fatal error is detected during the global analysis, or if the user changes either the modular structure of the program (requiring a corresponding change in the module list) or the TPF. After obtaining an INIT run with no fatal errors, translation of individual modules may begin.

4.1.2 Translating Modules to Ada

One module may be translated to Ada per run of the Translator. When invoking the Translator in TRAN mode, the inputs required are the J73 module to be translated and the TPF. The global analysis performed during INIT will be updated if necessary, and an Ada translation will be output (with diagnostics). The user may re-translate a module for any of the following reasons:

a. The module was modified to correct a translation error;

b. The module was modified for algorithmic reasons;

c. The module references another module which was re-translated since the current module was last translated;

d. The user requires a repeat of an earlier translation to obtain additional output listings.

The Translator will issue diagnostics which advise the user of needed re-translations for cases a. and c. In some cases, modification of an individual module may require reinitialization (for example, when adding or deleting compool directives from a module).

4.2 System Logical Flow

The Translator system's logical flow is described by the Software Conversion Cycle (Figure 2-1) and by the chart of Figure 4-1. Further details are presented in Section 4.4.

```
                          INVOKE TRANSLATOR
                                  |
                                  |
           -------------          |        -------------
           | INITIALIZE |         |        | TRANSLATE |
      -----|    THE     |---- MODE? ---|      A      |-----
      |    | TRANSLATION|                |   MODULE    |    |
      |     -------------                -------------      |
      |                                                     |
 ANALYZE                                                 UPDATE
 EACH                                                    MODULE
 MODULE                                                  ANALYSIS
      |                                             (IF NECESSARY)
      |                                                     |
      |                                                     |
 OUTPUT                                                     |
 DIAGNOSTICS                                             TRANSLATE
      |                                                     A
      |                                                  MODULE TO
      |                                                    ADA
      |                                                     |
 OUTPUT                                                     |
 DICTIONARY                                              OUTPUT ADA
      |                                                  LISTING
      |                                                     |
      |                                                     |
      -------------------------------------------------------
                                  |
                                  |
                                EXIT
```

Figure 4-1: LOGICAL FLOW

4.3 System Data

The following paragraphs describe the inputs, outputs, and internal data used by the Translator.

4.3.1 Inputs

Four types of inputs are required by the Translator: command input, J73 source modules, a J73 module list, and the translation parameters.

4.3.1.1 Command Input

A user of the Translator must supply command input to specify the following items:

a. Mode (INIT or TRAN)
b. Options (output listing; analyze/translate)
c. File names or device names of inputs and outputs.

The options that may be requested include dictionary listings, listings of Ada source generated by the Translator (see 4.3.2.2), and diagnostic suppression. When invoking TRAN mode, the user may specify analysis only (for diagnostics), translation only (for translating a module which has not been modified since it was last analyzed), or both (default). The user must inform the Translator (via the operating system) of the (host-dependent) file or device names needed for a run of the Translator, including the names of the files/devices to be used for input and output J73 and Ada modules, the TPF, the module list, and the dictionary.

4.3.1.2 J73 Source Input

To initialize the Translator, the user must provide the source code of the entire J73 program to be translated. To translate an individual module, the user must provide the source code for any portion of the program that is separately compilable by a J73 compiler (i.e., a single file whose first line is START and whose last line is TERM).

All J73 code must be syntactically correct, which implies that it has been previously checked by either a compiler or a code auditor. Previous compilation or auditing of the J73 source code is not mandatory; however, because the Translator will not process syntactically incorrect input, reliable translation will result only from input that is absolutely free of syntax errors. Input which is syntactically correct but erroneous will, in general, have unpredicable results. Some specific instances of erroneous program translation are discussed in the Functional Description.

3.1.3 Module List Input

order to perform the global analysis of the J73 program during IT mode, the Translator must have a means of identifying the urce files of the program according to module type (compool, ogram, procedure, or copy). This information is input using the dule List. The Module List is a text file consisting of one cord (i.e., card image) for each J73 source file to be anslated. Each record has the the format

<filename> <type>

ere the filename is a host-dependent identifier and the type is ther "compool", "program", or "procedure", if the file contains parately compilable J73 source, or "copy", if it contains text ich is input by one or more modules using a COPY directive. The dule List enables the Translator to perform a top-down analysis the program without requiring the user to submit the dividual modules in J73 "compilation order".

3.1.4 Translation Parameter Input

e Translation Parameter File (TPF) is used to guide the anslation of J73 constructs whose mapping to Ada is either bitrary or indefinite. The content of the TPF is described in pendix 1. The TPF is a user-accessible text file; it may be dified before initialization of the Translator (see 4.1.1) and y be listed or copied anytime.

tries in the TPF for the control of listing formats and comment ocessing may be overridden by user command inputs for dividual Translator runs; other translation parameters must main constant throughout the translation of a program.

3.2 Outputs

e Translator produces three types of outputs: Ada source code anslated from J73, Ada source code generated by the Translator, d a program dictionary. Each of these outputs may either be ored in a file or sent to a device such as a terminal or inter.

4.3.2.1 Translated Ada Module Output

The major output of the Translator will be a listing of the Ada module produced by a run of the Translator. This listing will be appropriately formatted ("prettyprinted") to conform to standard programming practices, including indentation to exhibit nesting, matching of "begins" and "ends", and form feeds for modular units (i.e., a new procedure gets a new page). Comments from the input J73 program will be included in the Ada listing if requested by the user. Warning messages will clearly delimit any missing Ada code corresponding to untranslated J73. The listing may be output to either a hard copy device (printer) for human inspection or to a file (disk or tape) for storage.

4.3.2.2 Generated Ada Module Output

A number of J73 constructs, including predefined types, certain intrinsic functions, and certain type converisons, have no exact Ada equivalent. Each such construct is translated to a type or function which is declared in a special Ada package called "J73_predefined_package". This package is derived by the Translator during INIT mode, updated as necessary during TRAN mode, and output as Ada source code upon user command. The rationale for the generation of J73_predefined_package is discussed in section 3.2.2.1 of the Functional Description; the specific contents of the package are described in sections 3.2.2, 3.2.3.1, and 3.2.5 of the Functional Description.

4.3.2.3 Program Dictionary Output

For translation purposes, the Translator must keep an internal dictionary of the names of all modules and externals used in the program being translated. A listing of this dictionary may be output upon request of the user. It will contain the name of each library unit in the Ada translation, as well as external names listed according to which library unit contains either a definition of or a reference to each external.

4.3.3 Data Base

This section defines the internal data base elements used by the Translator. The principle structures are the Module Table (one for the entire J73 program) and the symbol table, parse tree, and DIANA tree (one each for every J73 module).

4.3.3.1 Module Table

The program module table (ModTab) is a global data base which is used to store information about modules and externals. ModTab is initialized, using the user-supplied Module List, to include an entry for each J73 source module that identifies each module as a compool, program, procedure, or function. As each module is analyzed (see 4.4.3), its ModTab entry is filled in with the following data:

a. Number of other modules referenced (by REF's, copy directives, and compool directives);

b. For each module REF'ed, a pointer to that module's ModTab entry;

c. For each compool which is selectively imported, a pointer to a list of selected names;

d. A pointer to the module's symbol table;

e. A pointer to the module's parse tree.

The information contained in ModTab permits the Translator to resolve module dependencies and external references, to manage the creation and replacement of symbol tables and parse trees, and to generate a program dictionary. The internal structure of ModTab is implementation dependent.

4.3.3.2 J73 Module Representation

Each J73 module is internally represented by a symbol table (SymTab) and a parse tree. These two structures contain the syntactic and semantic data which the Translator requires for the analysis and translation of individual J73 modules. The parse tree provides a basis for the translation to DIANA (see 4.4.5 and 4.4.6) that is much more efficient than the direct processing of card image source text would be. The SymTab, along with ModTab, serves as the primary data base used in the analysis of J73 modules (described in 4.4.3); it also doubles as the identifier-attribute portion of the DIANA tree, as described in the next section.

4.3.3.3 Intermediate Language

The intermediate form of the Ada module to be output by the Translator is a DIANA syntax tree. The DEF_ID nodes of the tree are implemented as pointers into SymTab, so that the attributes of each variable do not need to be stored redundantly in the tree. Semantic and code attributes which are irrelevant to the translation process (such as sm_contraint and cd_alignment) are omitted. Two structural attributes have been added: as_error_number, which contains the identifier of a diagnostic, and as_error_link, whose value is a pointer into the J73 parse tree. These attributes, whose values are set to zero in the absence of translation errors, permit straightforward generation of diagnostics by GEN (see 4.4.7). Aside from these modifications, the DIANA tree conforms to the DIANA Reference Manual (reference [m]).

4.3.3.4 Other Data Base Elements

The Translator executive (4.4.1) creates a parameter table (ParmTab) based on the TPF. Because the translation parameters are needed frequently throughout the translation process, ParmTab is structured in a manner that permits very efficient lookup of the parameter values.

The J73_predefined_package (4.3.3.2) is internally constructed as a DIANA tree. The tree is expanded by SPEC (4.4.5) during the translation of each module, and is converted to Ada source by GEN (4.4.7) upon user command.

Other data includes a file of diagnostic message text, a table of J73 and Ada reserved words and symbols, the J73 parser table, and a file of diagnostics generated during INIT mode.

4.4 Program Descriptions

The following paragraphs describe the major functional components of the Translator. The highest structural level is depicted in Figure 2-2; the next highest level is discussed in this section. Lower levels of the program structure will depend on the details of an actual implementation of the Translator.

4.4.1 EXEC

The entry point of the Translator is called EXEC. EXEC performs two functions: it contains all of the routines which comprise the interface between the Translator and its host operating system and it serves as the main executive of the rest of the program. If the Translator is implemented using overlays, EXEC will include the commands necessary to accomplish the overlays.

The processing performed by EXEC is shown in Figure 4-2. The parameter table (ParmTab) is constructed from the TPF; other variables are initialized based on user command inputs. EXEC then calls either INIT or TRAN, based on the mode selected by the user, and then calls LIST to complete the Translator run.

```
                                   Begin EXEC
                                        !
                                        !
                                        v
                                 ---------------
                                 ! INITIALIZE  !
                                 !    DATA     !
                                 !  INCLUDING  !
                                 !   PARMTAB   !
                                 ---------------
                                        !
                                        v
                                       / \
                                      /   \
                         INIT        /     \     TRAN
        ----------------------------/ MODE  \-----------------------
        !                           \   ?   /                      !
        !                            \     /                       !
        !                             \   /                        !
        v                              \ /                         v
   ===========                                               ============
   !  INIT   !                                               !   TRAN   !
   ===========                                               ============
        !                                                          !
        !                                                          !
        !                                                          !
        !                                                          !
        !                                                          !
        --------------------------------------------------------------
                                        !
                                        !
                                        v
                                   =========
                                   ! LIST  !
                                   =========
                                        !
                                        !
                                        v

                                      exit
```

Figure 4-2: EXEC

4.4.2 INIT

INIT is the main routine for controlling the Translator in INIT mode. The primary function of INIT is to submit individual J73 modules to ANALYZE (see 4.4.3) in an order which permits an efficient global analysis of the J73 program to be performed.

In attempting to analyze a J73 module whose context may include compool data, two approaches are possible. The first approach is to use a compool output file to store the results of an analysis of the compool. The compool output file can then be imported into the data base (i.e., symbol table) of the module which references it, before the module itself is analyzed. This approach is appropriate for compilation of J73 for two reasons:

a. J73 modules may be coded and compiled in the order in which they must be analyzed by the compiler.

b. To compile a module which imports a compool, the compiler needs access only to the appropriate compool output files (not to the compool source).

Since these reasons are not applicable to the task of translating a complete, previously written J73 program to Ada, a different approach has been devised for use by the Translator. During INIT mode, the Translator builds a global data base by analysing each source module in "compilation order":

a. First, "stand-alone" compools;
b. Then, compools which import other compools;
c. Finally, the program, procedure, and function modules.

This approach removes the need for compool input/output processing. All the source files are available to the Translator at once during INIT mode; using information in the module table (ModTab), the INIT routine derives a correct order of analysis and proceeds to call ANALYZE for each module, building tne required global data base without the help of either compool output files or of a user-controlled ordering. This is a major advantage: it frees the user of the Translator from the task of manually deriving an acceptable ordering (a difficult task for a 1000 module program!) and also eliminates the time and space that would have been consumed by the creation and use of compool output files.

Before submitting the J73 source modules to ANALYZE in the fashion described above, INIT creates ModTab from the uesr-supplied Module List. INIT will terminate the analysis process when ANALYZE detects a fatal error in a module. A diagram of INIT appears in Figure 4-3.

```
                          begin INIT
                               |
                               |
                               v
                          ----------
                          | CREATE |
                          | MODTAB |
                          ----------
                               |
                               |
                               v
                       ---------------
                       |   SELECT    |
                       |  A MODULE   |<----------
                       | FROM MODTAB |          |
                       ---------------          |
                               |                |
                               |                |
                               v                |
                          ===========           |
                          | ANALYZE |           |
                          ===========           |
                               |                |
                               |                |
                               v                |
                             /   \              |
                            /     \             |
            YES            / FATAL \            |
            --------------\ERRORS?/             |
            |              \     /              |
            |               \   /               |
            |             NO  |                 |
            |                 v                 |
            |               /   \               |
            |              / ANY \      YES     |
            |             /MODULES\-------------|
            |             \ LEFT  /
            |              \  ?  /
            |               \   /
            |             NO  |
            |                 v
            |
            ------------->  Exit
```

Figure 4-3: INIT

4.4.3 ANALYZE

The purpose of the ANALYZE routine is to perform an analysis of an individual J73 source module in the context of the entire J73 program being translated. To do this, it is required that all modules on which a given module depends have been previously analyzed (as discussed in the preceeding section). ANALYZE may be called during INIT mode, to perform the initial analysis of a module, or during TRAN mode, to update the analysis for a new translation.

Processing in ANALYZE occurs in two parts, as shown in Figure 4-4. The first part is a syntactic analysis, in which the J73 source module is converted to a SymTab and a parse tree. The second part is an updating of the internal data bases related to the module analysis.

Syntactic analysis involves three routines: a tokenizer, a parser, and an error detector. The tokenizer performs a table-driven lexical analysis of each J73 symbol. It returns a keyword token for each predefined J73 symbol, and a name token (i.e., a character string) for each user defined symbol. The name tokens reflect the translated spellings of the user defined symbols, permitting detection of name conflicts during the analysis. The parser expands the symbol table and parse tree to reflect the syntactic content of the module using a conventional bottom-up parse algorithm. The parser may be generated automatically using a commercially available parser-generator, as in [9], or may be manually coded. In either case, the parser will generate simple diagnostics for any J73 syntax errors; no extraordinary error recovery techniques are needed, since the J73 input is supposed to be syntactically correct. However, since the J73 code may contain untranslatable constructs, the error detector is called by the parser to detect problematical J73 constructs (see Appendix 1 of the Functional Description) and name conflicts, making an entry in a diagnostics file for each error detected. The syntactic analysis is depicted in Figure 4-5.

Upon detection of an irrecoverable error, such as a missing copy file, missing compool, or J73 syntax error, ANALYZE will delete the erroneous SymTab and parse tree created by the syntatic analysis. If no fatal errors are encountered, ModTab is searched to yield the names of all modules which reference the current module. The names of these modules and their corresponding source files are stored in a table for use by LIST. If ANALYZE was called to update a module's analysis (rather than initialize it), the final action taken is to delete the module's previously created SymTab and parse tree.

```
                          Begin ANALYZE
                                !
                                !
                                v
                          -------------
                          !  PERFORM   !
                          ! SYNTACTIC  !
                          ! ANALYSYS   !
                          -------------
                                !
                                !
                                v
                               / \
                              /   \
                             /     \
            YES             / FATAL \
      ----------------------\ERRORS?/
      !                      \     /
      !                       \   /
      !                        \ /
      !                     NO  !
      v                         !
---------------                 !
!    PURGE     !                !
! SYMTAB AND   !                !
! PARSE TREE   !                !
---------------                 !
      !                         v
      !                 ----------------
      !                 !    CREATE     !
      !                 !  DEPENDENT    !
      !                 !    MODULE     !
      !                 !     LIST      !
      !                 !---------------!
      !                 !    DELETE     !
      !                 !  OLD SYMTAB   !
      !                 !  AND PARSE    !
      !                 !     TREE      !
      !                 ----------------
      !                         !
      ------------------------->!
                                v
                               exit
```

Figure 4-4: ANALYZE

```
                 Begin SYNTACTIC ANALYSIS
                          |
                          |
                          v
                  ----------------
                  | TOKENIZE     |
                  | A STATEMENT  |<-----------------
                  ----------------                 |
                          |                        |
                          |                        |
                          v                        |
                  ----------------                 |
                  |  PARSE THE   |                 |
                  | TOKEN STRING |                 |
                  ----------------                 |
                          |                        |
                          |                        |
                          v                        |
             -------------------------             |
             |  DETECT POTENTIAL     |             |
             | TRANSLATION ERRORS    |             |
             -------------------------             |
                          |                        |
                          |                        |
                          v                        |
             -------------------------             |
             | UPDATE DIAGNOSTIC     |             |
             |        FILE           |             |
             -------------------------             |
                          |                        |
                          |                        |
                          v                        |
                         / \                       |
                        /   \                      |
                       /     \                     |
                      /       \                    |
                     /  MORE   \                   |
                    /STATEMENTS \___YES____________|
                    \    ?      /
                     \         /
                      \       /
                       \     /
                        \   /
                          |
                   NO     |
                          v
                         exit
```

Figure 4-5: SYNTACTIC ANALYSIS

4.4.4 TRAN

TRAN is the routine which controls the Translator during TRAN mode. As shown in Figure 4-6, TRAN is a simple executive whose function is to call other routines based on the analyze/translate option requested by the user (see Section 4.3.1.1).

The user may wish to have a module analyzed, to detect possible translation errors or name conflicts, without needing an actual Ada source output. This is analogous to running a module though a compiler with a "syntax only" option; the user may obtain "front-end" diagnostics without paying for "back-end" processing. In this case, TRAN will call ANALYZE and then return without any further processing. Conversely, the user may wish to translate a module which has not been modified since it was last analyzed. This situation occurs when

a. The module has not been translated or modified since Translator initialization; or

b. The user desires additional output listings for the existing version of a module.

In this case, TRAN bypasses the call to ANALYZE and calls the routines SPEC, BODY, and GEN to perform the translation based on a prior analysis of the module.

```
                          Begin TRAN
                              |
                              v
                          /       \
                         /         \
  -----YES--------------/TRANSLATE\
  |                     \  ONLY ? /
  |                      \       /
  |                       \     /
  |                    NO    |
  |                          v
  |                    =============
  |                    |  ANALYZE  |
  |                    =============
  |                          |
  |                          v
  |                        /   \
  |                       /     \
  |                      / FATAL \---YES--------
  |                      \ERRORS?/             |
  |                       \     /              |
  |                        \   /               |
  |                    NO    |                 |
  |                          v                 |
  |                        /   \               |
  |                       /     \              |
  |                      /ANALYZE\    YES      |
  |                      \ ONLY? /------------>|
  |                       \     /              |
  |                    NO  \   /               |
  ------------------------->|                  |
                            v                  |
                    =============              |
                    |    SPEC   |              |
                    =============              |
                            |                  |
                            v                  |
                    =============              |
                    |    BODY   |              |
                    =============              |
                            |                  |
                            v                  |
                    =============              |
                    |    GEN    |              |
                    =============              |
                            |-------------------
                            v
                          exit
```

Figure 4-6: TRAN

4.4.5 SPEC

SPEC is the first of two routines which translate the J73 parse tree created by ANALYZE into a DIANA tree which represents the Ada translation of a module. The output of SPEC is the portion of the DIANA tree needed to represent the specification (i.e., visible part) of the package into which the J73 module will be translated. The DIANA tree is completed by BODY, which is discussed in the next section.

The translation of the J73 parse tree to DIANA is based on a top-down traversal of the parse tree. SPEC ignores the portions of the parse tree which correspond to the Ada package or procedure body, while creating the DIANA tree for the package specification according to the mappings discussed in the Functional Description. This includes the nodes and associated attributes for the package's context specification (WITH and USE clauses), as well as for the declarations which form the package specification itself. SPEC also adds nodes to the J73_predefined_package DIANA tree as required.

4.4.6 BODY

The second pass over the J73 parse tree is made by BODY. For compools, which are translated to package specifications with no package body, the entire DIANA tree is created by SPEC; BODY produces no output. Conversely, a procedure containing no DEF declarations or STATIC declarations is processed in entirety by BODY, since it is translated to a procedure body (with no package specification). In the general case (translation of procedures which may include DEF or STATIC data), the two-pass process performed by SPEC and BODY permits translation of the J73 parse tree to DIANA in an efficient manner; a one-pass technique would involve reordering of the module's declarations and statements to separate the package specification part from the package body part, requiring more complex tree-processing algorithms.

.4.7 GEN

da source code is generated from the DIANA tree by the GEN outine. GEN is a tree-walking algorithm which creates source ext based on the guidelines discussed in references [d] and [m]. n particular, each node of the DIANA tree created by SPEC and ODY will include the lx_comments attribute as suggested in ppendix III of [m]. The value of this attribute may be filled in ith a reference to a Translator-generated comment in the case of node that represents a Translator-generated statement; therwise, the attribute will contain a reference to an original 73 source comment (possibly null). GEN will use the s_error_number and as_error_link attributes (defined in 4.3.3.3) o generate diagnostic messages and J73 source code in positions f the Ada source corresponding to translation errors.

he output of GEN is a text file which is used by LIST to produce n appropriately formatted output listing. If the user has equested a listing of J73_predefined_package, GEN will also reate a text file based on that package's DIANA tree.

.4.8 LIST

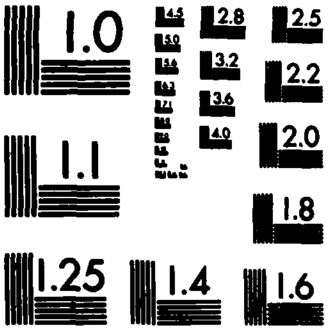
he listings output by the Translator are produced by LIST. Using he diagnostic files created by ANALYZE, the dictionary epresented by ModTab, and the Ada source files created by GEN, IST outputs prettyprinted reports requested by the user for each ranslator run.

AD-A120 472 JOVIAL (J73) TO ADA TRANSLATOR(U) PROPRIETARY SOFTWARE INC LOS ANGELES CA M J NEIMAN JUN 82 RADC-TR-82-175 F30602-81-C-0217 2/2

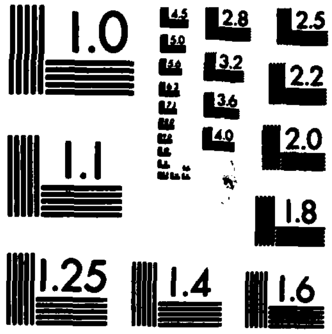
UNCLASSIFIED F/G 9/2 NL

END

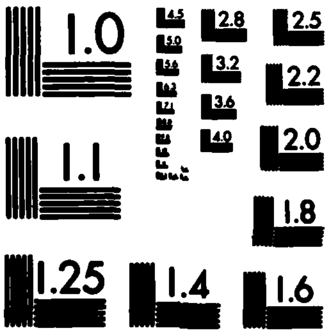
FILMED



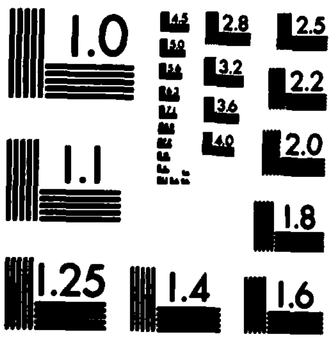
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



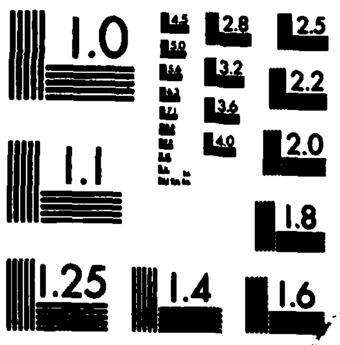
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

APPENDIX

Contents of the Translation Parameter File

The following table groups the translation parameters by class (comment translation, executable construct translation, J73 implementation parameters, control of output listings, name translation, and names of pragmas) and gives a brief description of the purpose of each parameter. (Note: T.G. means "Translator generated".)

PARAMETER CLASS	PARAMETER NUMBER	PURPOSE
COMMENTS	C1	Suppress original comments
	C2	Suppress T.G. comments
	C3	Start new line for an embedded comment
	C4	Format of comment for T.G. type declaration
EXECUTABLE	E1	Suppress calls to rounding routines
	E2	Suppress calls to truncation routines
	E3	Name of user supplied rounding function
	E4	Name of user supplied truncation function
	E5	Name of user supplied UNCHECKED_CONVERSION function
	E6	Convert IF...EXIT to EXIT WHEN...
	E7	Suppress copying of BY VAL parameters
	E8	Suppress copying of BY RES parameters
IMPLEMENTATION PARAMETERS	I1	Values of J73 implementation parameters
	.	
	.	
	.	
	I34	
LISTINGS	L1	Translation of tab stops
	L2	Translation of form feeds
	L3	Suppress upper case
	L4	Suppress lower case
	L5	Include J73 source in diagnostics
	L6	Suppress informational diagnostics
	L7	First column for unlabeled statements
	L8	Last column for code
	L9	Last column for comments

PARAMETER CLASS	PARAMETER NUMBER	PURPOSE
NAMES	N1	Translation of '
	N2	Translation of embedded $
	N3	Translation of leading $
	N4	Translation of names which are Ada reserved words
	N5	Spelling of T.G. type names
	N6	Spelling of T.G. function result names
	N7	Translation of status constant names
	N8	Maximum number of characters recognized
	N9	Spelling of T.G. package names
PRAGMAS	P1	Name of PRAGMA for contiguous allocation
	P2	Name of PRAGMA for overlayed allocation
	P3	Name of PRAGMA for !TRACE
	P4	Name of PRAGMA for !INTERFERENCE
	P5	Name of PRAGMA for !REDUCIBLE
	P6	Name of PRAGMA for !BASE
	P7	Name of PRAGMA for !ISBASE
	P8	Name of PRAGMA for !DROP
	P9	Name of PRAGMA for !LEFTRIGHT
	P10	Name of PRAGMA for !REARRANGE
	P11	Name of PRAGMA for !ORDER

GUIDELINES FOR TRANSLATION OF
JOVIAL (J73) PROGRAMS TO ADA

Prepared by:

Mark J. Neiman

PREFACE

This report was prepared as part of the JOVIAL (J73) to Ada Translator Investigation [6], a research study performed by Proprietary Software Systems for the Rome Air Development Center under contract number F30602-81-C-0217. Two other reports were prepared during the investigation: a Functonal Description [4], which defines the requirements to be met by a JOVIAL (J73) to Ada Translator, and a System/Subsystem Specification [5], which presents a top-level design for a Translator. The present report is intended to be read in the context of those two documents.

JOVIAL TO ADA TRANSLATOR INVESTIGATION
GUIDELINES FOR TRANSLATION

I. INTRODUCTION

A superficial inspection suggests that the languages JOVIAL (J73) and Ada, as defined by MIL-STD 1589B [2] and MIL-STD 1815 [3], are quite similar. Both languages feature separate compilation, strong typing, block structure, and compulsory data declarations. The IF statement, CASE statement, and loop constructs of the two languages are almost identical. Each language provides operations on fixed point and floating point data, in addition to integer and character operations. Ada is a much more powerful language than J73, since it includes many features for program control, modularization, and data description that are not found in J73. One might conclude that J73 is, in an informal sense, a functional "subset" of Ada, and that translating a J73 program to Ada should be a reasonably easy task.

Unfortunately, a closer analysis of the languages reveals a number of fundamental differences which render the translation task exceedingly complex. The semantics of data and type declarations is a case in point. In J73, the storage for a variable will be allocated statically (i.e., permanently) whenever the declaration of the variable so specifies; in Ada, storage is allocated by context (i.e., for the life of the module in which the variable's declaration appears). A J73 type is defined by a set of attributes, so that two distinctly declared types are considered to match if their atttributes match; two distinctly declared Ada types are always considered to be non-matching, even if their attributes are identical. The attributes of a type, in J73, are defined in terms of target machine representation (e.g., number of bits, physical record structure), while Ada requires only algorithmic attributes, such as range, error bounds, or logical record structure.

The two languages contain several major differences in the semantics of executable (run-time) constructs. J73 permits conversions between any two types, while Ada prohibits conversions between any two types which are not closely related. Linked structures may be created in a J73 program using untyped pointers to reference named (declared) objects; Ada allows only typed pointers, which may reference only anonymous objects. The semantics of parameter passing are defined in terms of binding mechanisms (value, reference, or result) in J73; Ada defines only the effect of a parameter binding (input or output), while carefully avoiding any specification of binding mechanisms. A J73 procedure may be prematurely terminated using one of two constructs ("GOTO <statement_name_parameter>" or the ABORT statement) which are nothing but global GOTO's; Ada permits only a well-structured mechanism (the raising of exceptions) to exit from a procedure prematurely.

The incompatibilities between J73 and Ada go beyond the semantic differences between their individual constructs. The languages have dissimilar requirements pertaining to the order of compilation of modules. An Ada compiler must have access to global knowledge of external names, whereas J73 externals are not resolved until link time. Both languages have a macro-definition/expansion facility, but J73 allows full, text-oriented macro substitutions, while Ada permits only procedure definitions (generics) in its macros.

The major differences between J73 and Ada are summarized in the table below. These differences, plus many smaller dissimilarities, cause the translation of a J73 program to Ada to be exceedingly difficult, whether the translation is done manually or with the aid of an automated system (a Translator). The portion of the translation task which can be automated is discussed in detail in the Translator Functional Description. A discussion of the portion which requires manual translation is given in the next two sections, followed by a section containing some guidelines for achieving "cleaner" translations.

Summary of J73/Ada Incompatibilities

FEATURE	J73	ADA
Static and external allocation	Explicit	By context
Type matching	Equivalent types always match	Distinct types do not match, even if equivalent
Attributes of types	Target machine oriented	Algorithm oriented
Order in which modules must be compiled	Imposed only on compool dependencies	Imposed on all modules
Type conversions	Permitted between any two types	Permitted between two closely related types
Relationship between pointers and pointed-to data objects	Untyped pointers, named data	Typed pointers, anonymous data objects
Macro-substitutions	Text-oriented	Procedure-oriented
Parameter passing	Defined by mechanism	Defined by effect
Abnormal termination of procedures	Unstructured GOTO and ABORT	Highly structured RAISE and EXCEPTION
Resolution of externals and parameter matching	Link-time	Compile-time

II. CLASSIFICATION OF PROBLEMATICAL CONSTRUCTS

Despite all the fundamental differences between J73 and Ada, there is probably no such thing as an untranslatable construct. Given enough analysis, any J73 program can be converted to Ada, FORTRAN, assembly language, or virtually any language which is intended for use on a conventional (von Neumann) computer. Unfortunately, the analysis and synthesis (i.e., reprogramming) required to translate certain J73 constructs to Ada automatically would be unreasonably expensive, given the state of the art of automatic programming and language conversion. A cost effective strategy is to automate the bulk of the translation task and to detect and identify (automatically) portions of programs which require manual translation. With this approach, a Translator system with roughly the same complexity as a J73 compiler would perform most of the translation without human assistance, while flagging the constructs that it cannot handle properly. These problematical constructs would then be analyzed and translated manually, using techniques outlined in Section III. Before discussing the translation of specific problematical constructs, it is useful to define classes of constructs according to ease of translatability.

Most J73 constructs can be translated to Ada using techniques which may be automated by the approach discussed in the Translator System/Subsystem Specification. Such "class-one" constructs are translated using the mappings described in the Translator Functional Description. The resulting Ada program will be better (more efficient and/or more readable) if the use of certain class-one constructs is avoided or restricted (see Section IV).

Constructs whose translation to Ada cannot be automated cost-effectively ("problematical" constructs) fall into two classes. An instance of a problematical construct which may be replaced by a non-problematical J73 construct is described as "class-two". Such a construct may be manually converted to a class-one construct to facilitate automatic translation. Problematical constructs which are semantically orthogonal to the rest of the J73 language present the most difficult translation problems. These are called "class-three" constructs. Since a class-three construct cannot be converted to a class-one construct, the translation requires one of the following actions:

A1. Change the J73 program algorithmically to avoid using the construct;

A2. Translate the rest of the program to Ada and change the Ada program algorithmically to avoid using the construct;

A3. Translate the rest of the program to Ada and use a feature (possibly a non-standard one) of the Ada environment to accomplish the function of the class-three construct;

A4. Substitute direct code (assembly language or machine language) for the construct.

The actions listed above may, of course, be taken to translate class-two constructs as well as class-three constructs.

The mappings of the class-one constructs onto Ada, as discussed in the Functional Description, are intended to be automated (see the Translator System/Subsystem Specification), but may be performed manually; the validity of the mappings is independent of the means of implementation. The problematical constructs, class-two and class-three, must be hand-translated. There are three kinds of problematical constructs: data-oriented constructs (table, block, and overlay declarations), executable constructs (global GOTO's, ABORT's, and expression side effects), and compile-time functions.

III. TRANSLATION OF PROBLEMATICAL CONSTRUCTS

A. Data-Oriented Constructs

A J73 data (or data type) declaration may specify several kinds of data overlaps. For example, a specified table may contain items whose bit positions (within the table entry) overlap either partially or completely; a block may be made to overlap another block using an overlay declaration and an order directive; overlay declarations may be used to position several data objects in overlapping positions in memory. In attempting to translate these kinds of constructs to Ada, one must consider the purpose of the construct. A particular instance of a problematical data declaration may have one of several purposes:

P1. A "true overlay", in which the same bits of physical memory are used by more than one named data object.

P2. The allocation of storage for data objects in a specified order.

P3. The allocation of contiguous storage of data objects.

P4. The allocation of storage for data objects at a specified memory address.

P5. A "virtual overlay", in which two or more named data objects are declared to occupy overlapping bit positions in a table or a block, but the data structure is accessed as a variant record (i.e., only one of the overlapping objects physically exists in each record; the objects do not really overlap).

A person wishing to translate a problematical data declaration to Ada must analyze the construct in the context of its program and determine into which of these categories it falls.

A "true overlay" may be treated as a class-two construct. This is accomplished by using duplicate storage in lieu of overlayed storage; instead of declaring one object to overlay the other, one may declare the objects as separately stored data. In the remainder of the program, each statement that changes one of the objects must be followed by a new statement that changes the other object in the same way. For example, a program of the form

```
ITEM ii1...
ITEM ii2...
OVERLAY ii1:ii2;
  .
  .
  .
ii1 = ...       "assignment to ii1 also assigns ii2"
```

is changed to

```
ITEM ii1...
ITEM ii2...
  .
  .
  .
ii1 = ...       "assigns only ii1"
ii2 = ii1;      "assigns ii2"
```

This technique has two major disadvantages. First, it is applicable only to "cleanly" overlayed objects - objects which are partially overlapped (such as table items) could not be recoded in this manner. Second, the resulting program is highly inefficient; twice as much storage is needed for the separately allocated objects and twice as many assignment statement statements are executed during the program. Because of these disadvantages, "true overlay" constructs should, in most applications, be treated as class-three constructs. An implementation of Ada may (optionally) provide an overlay construct, allowing action A3 to be used. If an overlay feature is not available, algorithmic changes (actions A1 or A2) are required.

A P5 construct ("virtual overlay") can be effectively translated using action A3. The technique is illustrated by the following example:

```
TABLE building (100)... "Table of data about two kinds of
                        buildings: home and business...
                        one entry per building"
BEGIN
   "The following items are used for all buildings:"
   ITEM zipcode U 17    POS (0,0);
   ITEM kind STATUS (v(home), v(business)) POS (17,0);

   "The following item is used for business buildings only:"
   ITEM name C 10       POS(0,1);  "name of business"

   "The following items are used for homes only:"
   ITEM bedrooms U 3    POS(0,1);  "number of bedrooms"
   ITEM baths U 2       POS(3,1);  "number of baths"
     .
     .
     .
END
```

is data structure, the items "bedrooms" and "baths" do not ly overlap "name". Instead, the item "kind" is used as a minant to select one of two alternate structures for each entry. This is semantically equivalent to a variant record . If the table declaration is translated to

```
 building_kind IS (home,business);
 building_type (kind: building_kind) IS
ECORD
  zipcode:  U17_type;
  CASE kind IS
      WHEN business => name: C10_type;
      WHEN home     => bedrooms: U3_type;
                       baths: U2_type;
  END CASE;
ND RECORD;
ding:  ARRAY (0...100) OF building_type;
```

assignments to all the items within an entry of "building" made using aggregate notation. Thus, the statements

```
ode(22) = 13411;
(22) = 2;
ooms(22) = 4;
```

anslated to

```
ding(22) := (home,13411,4,2); -- positional record aggregate
```

uivalently,

```
ding(22) := (kind => home, zipcode => 13411, baths => 2,
             bedrooms => 4);  -- named component aggregate
```

ecord aggregate used in the assignment includes a value for (the discriminant of the record), whether one uses the onal notation or the named component notation.

ignment to an individual item

```
s(22) = 2;
```

nslated to

```
ding (home)(22) := 2;
```

ich the discriminant is given on the left hand side and a l (rather than an aggregate) is given on the right hand

The use of a variant record for this kind of translation results in Ada code which is both efficient and semantically equivalent to the original algorithm of the J73 code.

When an overlay declaration is used for purpose P4 rather than for a "true overlay", it may be translated to an Ada address specification. For example,

```
OVERLAY POS (4FFF): block1 ;
```

is equivalent to

```
FOR block1 USE AT 16#4FFF# ;
```

Overlay declarations, block declarations, and order directives which are used for purposes P2 and P3 are not covered by the semantics of Ada as given by the language standard. Translation of such constructs may be achieved by action A3 if the Ada compiler/environment to be used offers optional features for overlaying or ordering of storage allocation. Otherwise, major algorithmic changes will be required.

B. Executable Constructs

When an ABORT statement is executed, the J73 procedure currently executing will terminate (return without setting any value or result parameters), and execution proceeds at the statement whose label appeared in the abort-phrase of the most recent procedure call statement which included an abort-phrase. If there were intermediate procedure calls without abort-phrases, then those intermediate procedures are also terminated; if no procedure calls included an abort-phrase, a STOP is executed. The difference between the ABORT statement and the Ada RAISE statement is that the ABORT may result in a transfer to any part of the procedure which handles the ABORT. The exception handler which is invoked by a RAISE statement must appear at the end of the procedure in which it appears; the handler acts as a substitute for the remainder of the calling procedure. In effect, a J73 ABORT is handled by executing an unrestricted GOTO within the calling procedure, while Ada permits a procedure termination to be handled only by a structured exit from the calling procedure.

The J73 statement name parameter is used to terminate a procedure with an unrestricted GOTO in the same manner as the ABORT, but at one level of procedure calls rather than any number of levels. The statement, "GOTO <statement_name_parameter>" is, therefore, a special case of the ABORT statement; the two constructs share the same class-three incompatibility with Ada.

Two techniques are available for translating a program which contains either of these constructs. The first involves an A4 action: replace the ABORTs and global GOTOs with calls to machine-level runtime routines, effecting the handling of procedure termination at the target-machine level rather than the high-order language level. The second technique is an algorithmic change (A2) which restructures the calling procedure, placing the logic which handles the ABORT or GOTO at the end of the procedure. Once this restructuring has been accomplished, the abort-phrase or statement name parameter is replaced by an exception declaration; the end-of-procedure logic is labeled as an exception handler; and the ABORT or GOTO is replaced by a RAISE statement. This technique of processing procedure terminations may be used for any number of statement name parameters or abort-phrase values, since multiple exceptions may be defined within the same procedure. The programming of exceptions is discussed in detail in the Ada language standard (in particular, see Section 11.4.1); a lucid discussion of the definition and propagation of exceptions may be found in Chapter 10 of Barnes [1].

The J73 language guarantees that the right-hand side of an assignment statement will be evaluated before the left hand side, and that function arguments and table indices will be evaluated, left to right, before any expressions or assignments are performed. This means that the statement

```
xx = func1("expression 1") + func2("expression 2");
```

may have a different effect than the statement

```
xx = func2("expression 2") + func1("expression 1");
```

if the evaluation of expression 1 causes a change (side effect) in the value of expression 2. A J73 program may actually rely on this effect; an Ada program may not. Beside avoiding such dubious programming practices, a programmer may remove order-of-evaluation dependencies from expressions and assignments by breaking up the expressions into separate statements. For example, if the preceeding assignment statement needs to have expression 1 evaluated first, then

```
xx = func1("expression 1");
xx = xx + func2("expression 2");
```

may be substituted for the original statement. This technique may be applied as either a J73 modification (treating it as a class-two problem) or as a change to the Ada translation of the program. In either case, the side effect dependencies must be detected and eliminated manually.

C. Compile-Time Functions

Because Ada lacks a text-oriented macro-capability, the DEFINE calls in a J73 program must be expanded at translation time. Therefore, the DEFINE declaration and the !LISTINV, !LISTEXP, and !LISTBOTH directives are simply discarded rather than translated. Other J73 directives which have no Ada equivalents may be translated only if the Ada environment to be used contains optional features which correspond to the J73 directives. In particular, the !TRACE directive will be implemented, in some form, in every Ada environment. Other directives (!REDUCIBLE, !BASE, !ISBASE, !DROP, !INTERFERENCE, !LEFTRIGHT, and !REARRANGE) have no runtime semantic effect; they simply aid the J73 compiler in performing certain code optimizations. Since these optimizations do not change the semantics of the program, and since Ada compilers are expected to perform subtle code optimizations without the assistance of such directives, it is likely that the deletion of these directives from a translated program will have no detrimental effect.

IV. PROGRAMMING GUIDELINES

In the preceeding sections, the translation of problematical constructs in existing J73 programs was discussed. If a J73 program is to be written with translation to Ada planned for the future, the program should avoid the use of all the problematical constructs. A J73 program containing only class-one constructs will be (relatively!) simple to translate; in fact, it will be automatically translatable. However, the J73 programmer can go beyond merely writing a non-problematical program. The Ada program that is produced by the translation process, whether manually or automatically, will be of significantly higher quality if the following guidelines are observed by the J73 programmer:

1. Do not use untyped pointers. Every pointer declaration should include a specified type, so that translation to access types is simplified.

2. Avoid conversions between unrelated types. Ada does not permit such conversions, except by use of the generic function UNCHECKED_CONVERSION, which is somewhat cumbersome to instantiate and call for every type of conversion.

3. Do not use names containing more than one consecutive $ or '. This practice will avoid the generation of awkward names using underscores in the Ada program. In fact, the names in the translated program will be much cleaner if the J73 names use either $ or ', but not both.

4. Limit the length of names to much less than the 32 characters permitted by J73. Many translation functions require the generation of type names based on adding an extension to an object name (or generation of package names by adding an extension to a procedure name), which may result in excessively long identifiers.

5. Do not use the FALLTHRU construct. Its translation is both awkward and inefficient.

6. Avoid loop statements with by-clauses or then-clauses which result in a loop increment of other than one. Virtually any function that requires a loop can be coded using either a FOR loop with an increment of one or a WHILE <condition> loop, both of which have simple and efficient Ada translations; loops with increments not equal to one can be translated, but not as cleanly.

7. Avoid elaborate DEFINE usage. DEFINEs will be lost in the translation process.

8. Declare global data in compools. Individual data declared as externals in procedures results in a much more complex translation. Similarly, static data should be declared in compools or in the main program, not in procedures.

9. Keep table structures as simple as possible. Programs which use parallel, packed, or variable entry tables will be much harder to translate to Ada than programs which use straightforward tables.

10. Include detailed comments about non-trivial data structures. Tables, blocks, and the code which accesses them can be translated much more easily (and tested much more reliably) if the personnel doing the translating and testing understand the purpose of the data structures.

11. Include detailed comments about pointer usage. Ada features very powerful instructions for dynamic allocation and access of linked data structures. These features may be exploited by manually recoding portions of a program (after translation) in Ada; a direct translation will not make efficient use of these features. This process will be facilitated by the liberal use of comments.

12. Avoid GOTOs and deeply nested procedures. This will improve the readability and maintainability of the program in both J73 and Ada versions.

V. CONCLUSIONS AND RECOMMENDATIONS

Many embedded software systems which are currently coded (or being coded) in J73 are to be used and maintained in the mid 1980's and beyond. Such systems should be considered as candidates for translation to Ada. The modification, enhancement, and repair of embedded systems will be much more economical if the (predicted) benefits of the Ada Program are exploited; the Ada Standardization Program guarantees that these benefits will be available only for Ada-coded systems. As shown by the Functional Description and the System/Subsystem Specification, a Translator can be implemented to perform the vast majority of the conversion to Ada of a large, realtime program. The translation process must not be considered to be "cookbook" in nature; even a well-designed Translator system will be unable to produce a flyable Ada program. The translation of problematical J73 constructs, as well as testing and integration of the Ada program, will require highly skilled personnel, whether or not a Translator is used. However, the total labor costs of producing a flyable program will be greatly reduced if such a tool is available.

Acknowledgement

The author is indebted to the following persons for major contributions to the Translator Investigation: Lonnie Brownell, Joel Fleiss, Richard Gilinsky, Guy Phillips, and Dale Rankin. Thanks also are due to Irene Evans for editing and word processing of the Functional Description, the System/Subsystem Specification, and this Report.

References:

[1] J.G.P Barnes, Programming_in_Ada, (Pre-publication copy, 1981).

[2] Mil-Std-1589B: JOVIAL_(J73), June, 1980.

[3] Mil-Std-1815: Reference_Manual_for_the_Ada_Programming Language, December 1980.

[4] M.J. Neiman, Functional_Description_for_the_JOVIAL_(J73)_to Ada_Translator, February 1982.

[5] M.J. Neiman, System/Subsystem_Specification_for_the_JOVIAL (J73)_to_Ada_Translator, March 1982.

[6] Statement of Work, contract F30602-81-C-0127, RADC/PSS, June 1981.